SCO® UNIX® Operating System

User's Guide

SCO®
OPEN SYSTEMS SOFTWARE

SCO® UNIX® Operating System

User's Guide

SCO®
OPEN SYSTEMS SOFTWARE

Date: 31 January 1992
Document Version: 3.2.4C

Chapter 1

Introduction 1

About this guide 1

Documentation conventions 2

Locating manual pages 4

Using online manual pages 4

The keyboard 6

Chapter 2

Creating, editing, and printing files 7

A vi tutorial 8

Entering the editor 8

Inserting text 9

Repeating a command 10

Undoing a command 10

Moving the cursor 11

Deleting text 12

Searching for a pattern 14

Searching and replacing 16

Leaving vi 17

Adding text from another file 17

Leaving vi temporarily 18

Changing your display 18

Canceling an editing session 19

Editing tasks 20

How to enter the editor 20

Moving the cursor 20

Moving around in a file 23

Inserting text before the cursor 23

Correcting typing mistakes 24

Opening a new line 24

Repeating the last insertion 24

Inserting text from other files 24

Inserting control characters into text 28

Joining and breaking lines 28

Deleting a character 28

Deleting a word 28

Deleting a line 29
Deleting an entire insertion 29
Deleting and replacing text 30
Moving text 32
Searching for patterns 35
Searching and replacing 36
Pattern matching 38
Undoing a command 40
Repeating a command 41
Leaving the editor 41
Editing a series of files 42
Editing a new file without leaving the editor 44
Leaving the editor temporarily 44
Performing a series of line-oriented commands 45
Finding out what file you are editing 45
Finding out what line you are on 46
Solving common problems 46
Setting up your vi environment 47
Setting the terminal type 48
Setting options with the set command 48
Displaying tabs and end-of-line 49
Ignoring case in search commands 50
Displaying line numbers 50
Printing the number of lines changed 50
Changing the terminal type 50
Shortening error messages 50
Turning off warnings 51
Permitting special characters in searches 51
Limiting searches 51
Turning on messages 51
Mapping keys 52
Abbreviating strings 52
Customizing your environment 53
Printing Files 53
Using lp 53
Using lp options 54
Canceling a print request 54
Finding out the status of a print request 55

Processing text files 56
Comparing files 56
Sorting files 56
Removing repeated lines in a file 57
Searching for patterns in a file 57
Counting words, lines, and characters 58
Summary 59
Summary of vi commands and variables 59
Summary of print commands 63
Summary of file processing utilities 63

Chapter 3
Communicating using mail **65**

Basic concepts 65
Mailboxes 66
Messages 66
Modes 67
Headers 68
Command syntax 69
Message lists 70
Using mail 71
Composing and sending a message 71
Reading your mail 72
Saving a message 73
Printing a message 75
Replying to a message 75
Deleting a message 75
Forwarding mail 76
Executing shell commands 76
Sending mail to remote sites 76
Leaving mail 77
Leaving compose mode temporarily 77
Setting up your environment 80
Creating mailing lists 80
Keeping mail in the system mailbox 81
The Cc: prompt 81
Listing messages in chronological order 81

Using advanced mail features 82
Using mail as a reminder service 82
Redirecting incoming messages to other folders 82
Sending automatic responses when on vacation 84
Forwarding your messages to another person 86
Converting mailboxes to the new MMDF format 86
Handling large amounts of mail 87
Summary 88

Chapter 4
Using disks and tapes *91*

Formatting disks and tapes 91
Formatting floppy disks 92
Formatting mini cartridge tapes 93
Using the tar command 93
Creating backups with tar 94
Listing the contents of backups 95
Extracting files from backups 96
Shorthand tar notation 98
Copying disks 99
Summary 100

Chapter 5
Managing processes *101*

Running jobs in the background 101
Checking the status of processes 102
Killing processes 104
Prioritizing processes 105
Scheduling your jobs 106
Executing programs automatically 106
Submitting your crontab file 108
Delaying program execution 108
Summary 112

Chapter 6

Communicating with other sites 113

Using UUCP 114
Transferring files with uucp 114
Transferring files with uuto 118
Executing commands with uux 120
Logging in to remote systems 121
Using ct 121
Using cu 123
Summary 126

Chapter 7

Using a secure system 127

Terminology 127
The security administrator 128
Login security 128
Logging in 128
What to do if you cannot log in 129
Changing your password 129
Using another account 131
Using commands on a trusted system 131
Authorizations 131
The auths command 133
Security for files in sticky directories 134
Recommended security practices 134
Password security 134
Good security habits 135
Logging in and out 135
File security 136
Data encryption — commands and descriptions 136
crypt — encode/decode files 137
Encrypting and decrypting with editors 138
Summary 139

Chapter 8
The Bourne shell 141

Basic concepts 142
How shells are created 142
Commands 142
How the shell finds commands 143
Generation of argument lists 143
Quoting mechanisms 144
Standard input and output 145
Diagnostic and other outputs 146
Command lines and pipelines 147
Command substitution 149
Shell variables 150
Positional parameters 150
User-defined variables 150
Predefined special variables 154
The shell state 155
Changing directories 155
The .profile file 156
Execution flags 156
A command's environment 157
Invoking the shell 158
Passing arguments to shell procedures 158
Controlling the flow of control 160
Using the if statement 161
Using the case statement 163
Conditional looping: while and until 163
Looping over a list: for 164
Loop control: break and continue 165
End-of-file and exit 166
Command grouping: parentheses and braces 166
Defining functions 167
Input/output redirection and control commands 168
Transfer between files: the dot command 169
Interrupt handling: trap 169
Special shell commands 173
Creation and organization of shell procedures 175
More about execution flags 177
Supporting commands and features 177
Conditional evaluation: test 177

Echoing arguments 179
Expression evaluation: expr 179
True and false 180
In-line input documents 180
Input / output redirection using file descriptors 180
Conditional substitution 181
Invocation flags 183
Effective and efficient shell programming 183
Number of processes generated 184
Number of data bytes accessed 186
Shortening directory searches 186
Directory-search order and the PATH variable 186
Good ways to set up directories 187
Shell procedure examples 187
Shell grammar 194

Chapter 9
The C shell *197*

Invoking the C shell 197
Using shell variables 198
Using the C shell history list 200
Using aliases 202
Redirecting input and output 204
Creating background and foreground jobs 204
Using built-in commands 205
Creating shell scripts 207
Using the argv variable 207
Substituting shell variables 208
Using expressions 210
A sample script 211
Using other control structures 214
Supplying input to commands 215
Catching interrupts 215
Using other features 216
Starting a loop at a terminal 216
Using braces with arguments 217
Substituting commands 218
Special characters 218

Chapter 10

The Korn shell 221

Overview 221
Setting up the Korn shell 222
Flags and options for ksh 223
Editing features 224
Using the vi built-in editor mode 225
Editing in vi input mode 225
Editing in control mode 226
Accessing commands in the history file 228
Displaying commands in the history file 228
Re-executing previous commands 229
Editing previous commands 229
Configuring the history file 230
Manipulating commands wider than the screen 230
Using expanded cd capabilities 231
Using job control 232
Referring to jobs 233
Using the ksh job control commands 234
Running jobs in the background 234
Moving background jobs to foreground 234
Moving foreground jobs to background 235
Displaying information about jobs 235
Terminating a background job 235
External job control facilities 236
Interpretation of commands 237
Alias expansion 238
Quote expansion and ksh 240
Parentheses 241
Tilde expansion 241
Programming the Korn shell 242
Arguments, parameters, and variables 242
Built-in commands 244
Differences from the Bourne shell 246
Functions 246
Substring parameter expansion 248
The select command 250
Pattern matching 251
Debugging support 251

Chapter 11
Manipulating text with sed 253

Using sed 254
Addresses 255
Functions 257
Whole-line oriented functions 257
Substitute functions 259
Input-output functions 261
Multiple input-line functions 262
Hold and get functions 263
Flow-of-control functions 264
Miscellaneous functions 264

Chapter 12
Simple programming with awk 265

Basic awk 266
Program structure 266
Running awk programs 266
Fields 267
Printing lines 268
Formatting awk output 269
Using simple patterns 269
Using simple actions 270
A handful of useful one-liners 271
Error messages 272
Patterns 273
BEGIN and END 273
Relational expressions 274
Regular expressions 275
Combining patterns 277
Pattern ranges 278
Actions 278
Built-in awk variables 278
Performing arithmetic 279
Using strings and string functions 282
Field variables 285
Number or string? 286

Control flow statements 287
Arrays 289
User-defined functions 291
Some lexical conventions 292
awk output 292
The print statement 292
Output separators 293
The printf statement 293
Output into files 295
Output into pipes 295
Input 296
Files and pipes 296
Input separators 297
Multi-line records 297
The getline function 297
Command-line arguments 300
Using awk with other commands and the shell 300
The system function 300
Co-operation with the shell 301
Example applications 302
Generating reports 303
Additional examples 305
awk summary 307
Command line 307
Patterns 308
Control flow statements 308
Input-Output 308
Functions 309
String functions 309
Arithmetic functions 309
Operators (Increasing precedence) 310
Regular expressions (Increasing precedence) 310
Built-in variables 311
Limits 311
Initialization, comparison, and type coercion 312

Chapter 13
Using DOS accessing utilities 315

Accessing DOS files ... 315
Copying groups of files ... 317
Using mounted DOS filesystems ... 318
Mounting a DOS filesystem ... 318
File and directory arguments ... 319
User configurable default file ... 319
Appearance of DOS files ... 320
Newline conversions with DOS utilities ... 321
Other restrictions ... 321
Summary ... 322

Appendix A
Sample shell startup files 323

The Bourne shell .profile ... 323
The Korn shell .profile and .kshrc ... 325
The C-shell .login and .cshrc ... 328

Chapter 1

Introduction

This guide explains many of the features of your UNIX system. It shows you how to manage your files and directories, save copies of files on backup media, use security features, and schedule your jobs. It provides information on several of the most useful UNIX system facilities, including **mail**(C), the **vi**(C) text editor, and UUCP system. In addition, the guide includes information on the three UNIX system shells: the Bourne shell, the Korn shell, and the C shell. Information is given on creating shell scripts, with descriptions on how to use the **sed**(C) stream editor and the **awk**(C) programming language.

About this guide

This guide is organized into the following chapters:

Chapter 1, "Introduction," provides an overview of the contents of this guide and gives a list of the notational conventions used throughout.

Chapter 2, "Creating, editing, and printing files," shows you how to use the UNIX system full screen editor, **vi**, to create files. This chapter also explains how to send files to the printer and cancel print jobs and includes information on the basic utilities you can use to process text files.

Chapter 3, "Communicating using mail," explains how to use the UNIX system electronic mail facility to communicate with other sites.

Chapter 4, "Using disks and tapes," shows you how to use floppy disks and tapes to back up files and directories.

Chapter 5, "Managing processes," demonstrates how to run jobs in the background. This chapter also shows you how to schedule or delay the execution of your jobs using the **cron**(C), **at**(C), and **batch**(C) utilities.

Chapter 6, "Communicating with other sites," explains how to transfer files between computer sites and how to execute commands on other computer sites.

Chapter 7, "Using a secure system," discusses the security features that might be in use at your site and how to work with them.

Chapter 8, "The Bourne shell," explains how to use the UNIX system Bourne shell, including how to create "shell scripts" to run under this shell.

Chapter 9, "The C shell," explains how to use the UNIX system C shell.

Chapter 10, "The Korn shell," explains how to use the Korn shell, and includes information about the differences between Bourne and Korn shell programming.

Chapter 11, "Manipulating text with sed," demonstrates how to use the **sed** stream editor to automate file editing.

Chapter 12, "Simple programming with awk," shows how to use **awk** to write simple programs that you can use to manipulate files and data.

Chapter 13, "Using the DOS accessing utilities," explains how to access DOS files indirectly using the DOS utilities, or directly using mounted DOS file systems.

Appendix A, "Sample shell startup files," contains sample listings and line-by-line explanations for the various shell startup files.

Documentation conventions

The following documentation conventions are used in this guide.

boldface Commands are shown in **boldface**. For example:

... use the **rm** command to remove files ...

UNIX system utilities or library routines are also shown in **boldface**. For example:

... use the **grep** utility to search files for a specified pattern ...

Literal user input is also shown in **boldface**. For example:

... to display the file itself, enter:

more /etc/termcap

and press ⟨Return⟩ ...

italics

Directories and filenames are shown in *italics*.

For example:

... most executable files are found in the */bin* directory ...

Emphasized words or phrases are also shown in *italics*. For example:

... the constant creation and removal of files creates a situation called *disk fragmentation* ...

References to book titles are also shown in *italics*.

For example:

... for information relating to system administration, refer to the *System Administrator's Guide* ...

bold italics

Placeholders are shown in ***bold italics***. A placeholder is a word which you must replace with an appropriate filename, number, or option. For example:

... to change directories, enter:

cd ***directory***

where ***directory*** is the directory you want to change to ...

`courier`

Screen displays and other output from the computer are shown in `courier`. For example:

```
The 1200 baud dialer is busy
Do you want to wait for dialer? (y for yes)
```

" "

Data values and field names are shown in "quotation" marks. For example:

... where ***x*** is "0" for a display adapter or "1" for a serial port ...

Quotation marks are also used for normal words used in a way particular to computing. For example:

... your local site is a "dial out" site ...

Document chapter names are also shown in quotations. For example:

... consult the "Communicating using mail" chapter of this guide ...

SMALL CAPITALS

Acronyms are shown in SMALL CAPITALS. For example:

... the name UUCP is an acronym for UNIX to UNIX Copy ...

SMALL BOLD CAPITALS	System parameters (definable system values, for example, the number of disk drives attached to the system) and environment variables (definable system information, for example, what type of terminal is being used) are shown in **SMALL BOLD CAPITALS**. For example: ... **$1** and **$2** refer to positional parameters used ... ... the preferable method for setting your terminal type is to assign the type to the **TERM** variable ...
⟨ ⟩	Names of keys are shown in ⟨angle brackets⟩. For example: ... press the ⟨Esc⟩ key to exit the current mode ...

Locating manual pages

When you use the command line you have direct access to utilities and data. Notice the form used for commands in this guide. Each command is printed in bold type, and each has a suffix to help you find more information about it.

Table 1.1 lists the locations of the manual pages for the commands with the indicated suffixes. To find out information about a command, note the letter or letters that appear in parentheses following the command, then look up the command in the appropriate reference book or guide. For example, the command **lpstat**(C) is defined in the **Commands**(C) section of the *User's Reference.*

Using online manual pages

If manual pages are installed on your system they may be viewed by typing:

man ***command***

where ***command*** is the command for which you want to see the manual page.

For example, to see the manual page for the **more** command, type:

man more

Some manual pages appear in more than one location (see the following table). To see all occurrences of a particular manual page, type:

man -a ***command***

For example, to see all the manual pages for the **hd** command, type:

man -a hd

To force the system to display a manual page for a particular location, type:

man *location command*

For example, to see the manual page for the **hd** command for the location **HW**, type:

man HW hd

Table 1-1 Manual page locations

Command Suffix	Book and Purpose
ADM	*System Administrator's Reference* - commands reserved for the exclusive use of system administrators
C	*User's Reference* - operating-system commands available to all users
CP	*Programmer's Reference* - programming commands used with the development system
DOS	*Programmer's Reference* - DOS routines used with the development system
F	*System Administrator's Reference* - (File Formats) description of system configuration files
FP	*Programmer's Reference* - (File Formats) description of system files and data structures
HW	*System Administrator's Reference* - information about hardware devices and device nodes
K	*Device Driver Writer's Guide* - routines provided in the kernel for writing device drivers
M	*User's Reference* - miscellaneous information used for access to devices, system maintenance, and communications
S	*Programmer's Reference* - system calls and library routines for C and assembly-language programming

NOTE The *Programmer's Reference* and *Device Driver Writer's Guide* are only supplied if the Development System is purchased.

The keyboard

Many keys and key combinations perform special actions on UNIX systems. These actions have names that may not correspond to the keytop labels on your keyboard. Table 1.2 shows which keys on a typical terminal correspond to special actions on UNIX systems. A list for your particular login device is in **keyboard**(HW). Many of these keys can be modified by the user; see **stty**(C).

Table 1-2 Special keys

UNIX Name	Action
⟨Return⟩	terminates a command line and initiates an action. This key is also called the ⟨ENTER⟩ key; the keytop may indicate a down-left arrow.
⟨Esc⟩	exits the current mode; for example, exits insert mode when in the editor **vi**. This is also known as the ⟨ESCAPE⟩ key.
⟨Del⟩	stops the current program, returning to the shell prompt. This key is also known as the ⟨INTERRUPT⟩ or the ⟨DELETE⟩ key.
⟨Bksp⟩	deletes the character to the left of the cursor. The keytop may show a large left arrow, as opposed to the small "cursor left" arrow.
⟨Ctrl⟩d	signals the end of input from the keyboard; exits current shell, or logs you out if the current shell is the login shell. This is not interchangeable with the ⟨BREAK⟩ key.
⟨Ctrl⟩h	deletes the first character to the left of the cursor. This is also called the ⟨ERASE⟩ key.
⟨Ctrl⟩q	restarts printing after it is stopped with ⟨Ctrl⟩s.
⟨Ctrl⟩s	stops printing at the standard output device, such as a terminal. This keystroke does not stop the program.
⟨Ctrl⟩u	deletes all characters on the current line. This is also called the ⟨KILL⟩ key.
⟨Ctrl⟩\	quits current command, creates a *core* file. This is also called the ⟨QUIT⟩ key. (Use of this keystroke is recommended for debugging only; see **core**(F).)

Chapter 2
Creating, editing, and printing files

Any ASCII text file, such as a program or document, can be created and modified using a text editor. There are two text editors available on UNIX systems, **ed**(C) and **vi**(C). (For information on **ed**, see the **ed**(C) manual page.)

The **vi** command (which stands for "visual") combines line-oriented and screen-oriented features into a powerful set of text editing operations that satisfy any text editing need.

This chapter explains how to create, edit, print, and process text files on your UNIX system. It contains the following sections:

- A **vi** tutorial that gives you some hands-on experience creating and editing files. This section introduces the basic concepts you must be familiar with before you can really learn to use **vi**, and shows you how to perform simple editing functions.
- A reference section that explains how to perform specific editing tasks.
- A section covering some common problems and how to resolve them.
- A description of how to print the files you create with **vi**.
- An explanation of some common utilities for processing text files.
- A description of how to set up your **vi** environment and how to set optional features.
- A summary of the **vi** commands covered in this chapter.

Because **vi** is such a powerful editor, it has many more commands than you can learn at one sitting. If you have not used a text editor before, the best approach is to become thoroughly comfortable with the concepts and operations presented in the tutorial section, then refer to the second part for specific tasks you need to perform. All the steps needed to perform a given task are explained in each section, so some information is repeated several times.

When you are familiar with the basic **vi** commands you can easily learn how to use the more advanced features.

If you have used a text editor before, you might want to turn directly to the task-oriented part of this chapter. Begin by learning the features you use most often.

This chapter covers the basic text editing features of **vi**. If you are an experienced user of **vi** you might prefer to use the **vi**(C) manual page in the *User's Reference* instead of this chapter. **vi**(C) explains more advanced topics and features related to editing programs.

A vi tutorial

The following tutorial gives you hands-on experience using **vi**, and introduces some basic concepts that you must understand before you can learn more advanced features. This section explains how to enter and exit the editor, insert and delete text, search for patterns and replace them, and how to insert text from other files. This tutorial should take one hour. Remember that the best way to learn **vi** is to actually use it, so don't be afraid to experiment.

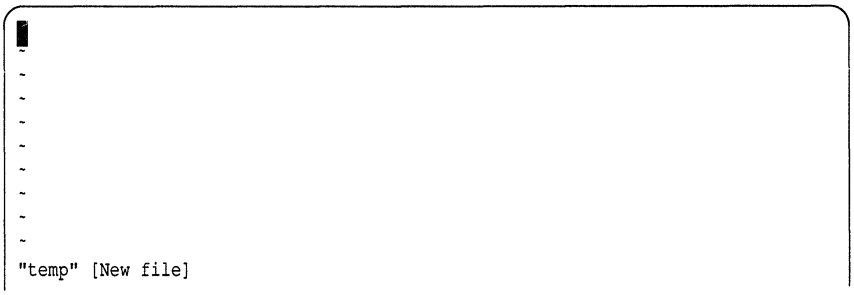
Before you start the tutorial, make sure that your terminal has been properly set up. See the section "Setting the terminal type," later in this chapter for more information about setting up your terminal for use with **vi**.

Entering the editor

To enter the editor and create a file named *temp*, enter:

```
vi temp
```

Your screen looks like this:



```
█
~
~
~
~
~
~
~
~
~
"temp" [New file]
```

Note that the examples in this chapter show a twelve-line screen to save space. In reality, **vi** uses whatever size screen you have.

You are initially editing a copy of the file. The file itself is not altered until you save it. Saving a file is explained later in the tutorial.

The top line of your display is the only line in the file and is marked by the cursor, shown above as █. In this chapter, when the cursor is on a character that character appears in reverse-video font.

The line containing the cursor is called the *current line.* The lines containing tildes are not part of the file; they indicate lines on the screen only, not real lines in the file.

Inserting text

To begin, create some text in the *temp* file by using the Insert (i) command. To do this, press **i**. Next, enter the following five lines to give yourself some text to experiment with; press ⟨Return⟩ at the end of each line. If you make a mistake, use the ⟨Bksp⟩ key to erase the error and enter the word again.

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
█
~
~
~
~
~
```

Press the ⟨Esc⟩ key when you are finished.

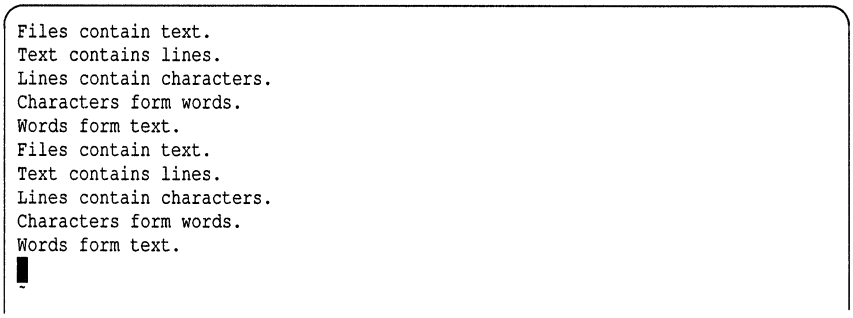
Like most **vi** commands, the **i** command does not appear on your screen. The command itself switches you from Command mode to Insert mode.

When you are in Insert mode, every character you enter is displayed on the screen. In Command mode, the characters you enter are not placed in the file as text; they are interpreted as commands to be executed on the file. If you are not certain which mode you are in, press ⟨Esc⟩ until you hear the terminal bell. When you hear the bell, you are in Command mode.

Once in Insert mode, **vi** inserts the characters that you enter into the file; **vi** does *not* interpret them as **vi** commands. Always press ⟨Esc⟩ to exit Insert mode and re-enter Command mode. This switching between modes occurs often in **vi**, and it is important to get used to it now.

Repeating a command

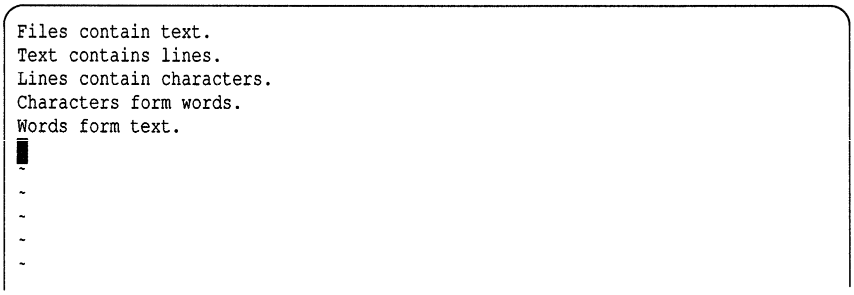
Next comes a command that you use frequently in **vi**: the Repeat command. The Repeat command repeats the most recent Insert or Delete command. Since we have just executed an Insert command, the Repeat command repeats the insertion, duplicating the inserted text. The Repeat command is executed by entering a period (.) or "dot." So, to add five more lines of text, enter " . ". The Repeat command is repeated relative to the location of the cursor and inserts text *below* the current line. (Remember, the current line is always the line containing the cursor.) After you enter dot (.), your screen looks like this:



```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
█
~
```

Undoing a command

Another command which is very useful (and which you need often in the beginning) is the Undo (**u**) command. Press **u** and notice that the five lines you just finished inserting are deleted or "undone."



```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
█
~
~
~
~
~
```

Now enter **u** again, and the five lines are reinserted! This undo feature can be very useful in recovering from inadvertent deletions or insertions.

Moving the cursor

Now, let's learn how to move the cursor around on the screen. In addition to the arrow keys, the following letter keys also control the cursor:

h	Left
l	Right
k	Up
j	Down

The letter keys are chosen because of their relative positions on the keyboard. Remember that the cursor movement keys only work in Command mode.

Try moving the cursor using these keys. (First make sure you are in Command mode by pressing the ⟨Esc⟩ key until you hear the bell.) Then, enter the **H** command to place the cursor in the upper left corner of the screen. Then enter the **L** command to move to the lowest line on the screen. (Note that case is significant in our example: **L** moves to the lowest line on the screen; while **l** moves the cursor forward one character.) Next, try moving the cursor to the last line in the file with the goto command, **G.** If you enter **2G,** the cursor moves to the beginning of the second line in the file; if you have a 10,000 line file, and enter **8888G,** the cursor goes to the beginning of line 8888. (If you have a 600 line file and enter **800G** the cursor does not move.)

These cursor movement commands should allow you to move around well enough for this tutorial. Other cursor movement commands you might want to try out are:

w	Moves forward a word
b	Backs up a word
0	Moves to the beginning of a line
$	Moves to the end of a line

You can move through many lines quickly with the scrolling commands:

⟨Ctrl⟩u	Scrolls up 1/2 screen
⟨Ctrl⟩d	Scrolls down 1/2 screen
⟨Ctrl⟩f	Scrolls forward one screenful
⟨Ctrl⟩b	Scrolls backward one screenful

Deleting text

Now that we know how to insert and create text, and how to move around within the file, we are ready to delete text. Many Delete commands can be combined with cursor movement commands, as explained below. The most common Delete commands are:

dd	Deletes the current line (the line the cursor is on), regardless of the location of the cursor in the line.
dw	Deletes the word above the cursor. If the cursor is in the middle of the word, deletes from the cursor to the end of the word.
x	Deletes the character above the cursor.
d$	Deletes from the cursor to the end of the line.
D	Deletes from the cursor to the end of the line.
d0	Deletes from the cursor to the start of the line.
.	Repeats the last change. (Use this only if your last command was a deletion.)

To learn how all these commands work, let's delete various parts of the tutorial file. To begin, press ⟨Esc⟩ to make sure you are in Command mode, then move to the first line of the file by entering:

1G

At first, your file looks like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
```

To delete the first line, enter **dd**. Your file now looks like this:

```
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

Delete the word the cursor is sitting on by entering **dw**. After deleting, your file looks like this:

```
contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

You can quickly delete the character above the cursor by pressing **x**. This leaves:

```
ontains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

Now enter a **w** command to move your cursor to the beginning of the word *lines* on the first line. Then, to delete to the end of the line, enter **d$**. Your file looks like this:

```
ontains█
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

To delete all the characters on the line *before* the cursor, enter **d0**. This leaves a single space on the line:

```
█
Lines contain characters.
Files contain text.
Text contains lines.
Characters form words.
Words form text.
Lines contain characters.
Characters form words.
Words form text.
~
~
```

For review, let's restore the first two lines of the file.

Press **i** to enter Insert mode, then enter:

> **Files contain text.**
> **Text contains lines.**

Press ⟨Esc⟩ to go back to Command mode.

Searching for a pattern

You can search forward for a pattern of characters by entering a slash (/) followed by the pattern you are searching for, terminated by a ⟨Return⟩. For example, make sure you are in Command mode (press ⟨Esc⟩), then press **H** to move the cursor to the top of the screen. Now, enter:

> **/char**

Do not press ⟨Return⟩ yet. Your screen looks like this:

```


Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
/char
```

Press ⟨Return⟩. The cursor moves to the beginning of the word *characters* on line three. To search for the next occurrence of the pattern *char*, press **n** (as in "next".) This takes you to the beginning of the word *characters* on the eighth line. If you keep pressing **n**, **vi** searches past the end of the file, wraps around to the beginning, and again finds the *char* on line three.

Note that the slash character and the pattern that you are searching for appear at the bottom of the screen. This bottom line is the **vi** status line.

The *status line* appears at the bottom of the screen. It is used to display information, including patterns you are searching for, line-oriented commands (explained later in this tutorial), and error messages.

For example, to get status information about the file, press ⟨Ctrl⟩g. Your screen should look like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain [c]haracters.
Characters form words.
Words form text.
~
"temp" [Modified] line 4 of 10 --4%--
```

The status line on the bottom tells you the name of the file you are editing, whether it has been modified, the current line number, the number of lines in the file, and your location in the file as a percentage of the number of lines in the file. The status line disappears as you continue working.

Searching and replacing

Let's say you want to change all occurrences of *text* in the tutorial file to *documents*. Rather than search for *text*, then delete it and insert *documents*, you can do it all in one command. The commands you have learned so far have all been *screen-oriented*. Commands that perform more than one action (searching and replacing) are *line-oriented* commands.

Screen-oriented commands are executed at the location of the cursor. You do not need to tell the computer where to perform the operation; it takes place relative to the cursor. *Line-oriented* commands require you to specify an exact location (called an "address") where the operation is to take place. Screen-oriented commands are easy to enter, and provide immediate feedback; the change is displayed on the screen. Line-oriented commands are more complicated to enter, but they can be executed independent of the cursor, and in more than one place in a file at a time.

All line-oriented commands are preceded by a colon which acts as a prompt on the status line. Line-oriented commands themselves are entered on this line and terminated with a ⟨Return⟩.

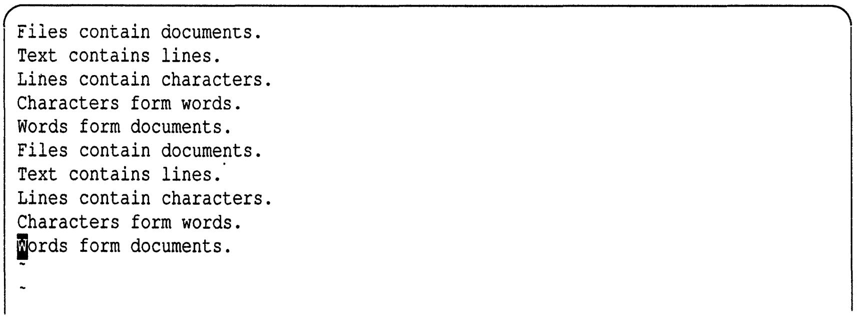
In this chapter, all instructions for line-oriented commands include the colon as part of the command.

To change *text* to *documents*, press ⟨Esc⟩ to make sure you are in Command mode, then enter:

:1,$s/text/documents/g

This command means "From the first line (1) to the end of the file ($), find *text* and replace it with *documents* (s/text/documents/) everywhere it occurs on each line (g)."

Press ⟨Return⟩. Your screen now looks like this:



```
Files contain documents.
Text contains lines.
Lines contain characters.
Characters form words.
Words form documents.
Files contain documents.
Text contains lines.
Lines contain characters.
Characters form words.
Words form documents.
~
~
```

Note that *Text* in lines two and eight was not changed. Case is significant in searches.

Just for practice, use the Undo command to change *documents* back to *text*. Press **u** and your screen now looks like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
```

Leaving vi

All of the editing you have been doing has affected a copy of the file, and *not* the file named *temp* that you specified when you invoked **vi**. To save the changes you have made, exit the editor and return to the UNIX shell, enter:

:x

Remember to press ⟨Return⟩. The name of the file, and the number of lines and characters it contains are displayed on the status line:

```
"temp" [New file] 10 lines, 214 characters
```

Then the UNIX system prompt appears.

Adding text from another file

In this section we will create a new file, and insert text into it from another file. First, create a new file named *practice* by entering:

vi practice

This file is empty. Let's copy the text from *temp* and put it in *practice* with the line-oriented Read command. Press ⟨Esc⟩ to make sure you are in Command mode, then enter:

:r temp

Your file should now look like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
```

The text from *temp* has been copied and put in the current file *practice.* There is an empty line at the top of the file. Move the cursor to the empty line and delete it with the **dd** command.

Leaving vi temporarily

vi allows you to execute commands outside of the file you are editing, such as **date**. To find out the date and time, enter:

:!date

This displays the date, then prompts you to press ⟨Return⟩ to re-enter Command mode. Go ahead and try it. Your screen should look similar to this:

```

Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
:!date
Mon Jan 9 16:33:37 PST 1985
[Press return to continue]
```

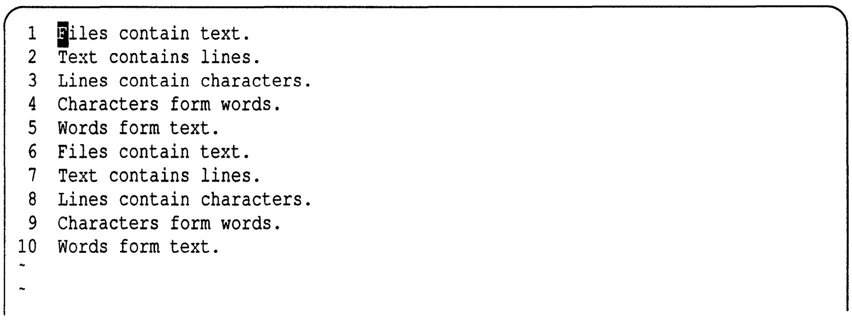
Changing your display

Besides the set of editing commands described above, there are a number of options that can be set either when you invoke **vi**, or later when editing. These options allow you to control editing parameters such as line number display, and whether or not case is significant in searches. In this section, we explain how to turn on line numbering, and how to look at the current option settings.

To turn on automatic line numbering, enter:

:set number

Press ⟨Return⟩. **vi** redraws your screen, and line numbers appear to the left of the text. Your screen now looks like this:



```
 1  Files contain text.
 2  Text contains lines.
 3  Lines contain characters.
 4  Characters form words.
 5  Words form text.
 6  Files contain text.
 7  Text contains lines.
 8  Lines contain characters.
 9  Characters form words.
10  Words form text.
~
~
```

You can get a complete list of the available options by entering:

:set all

Setting these options is described in the section "Setting up your vi environment," but it is important that you be aware of their existence. Depending on what you are working on, and your own preferences, you might want to alter the default settings for many of these options.

Canceling an editing session

Finally, to exit **vi** without saving the file *practice*, enter:

:q!

This cancels all the changes you have made to *practice* and, because it is a new file, deletes it. The UNIX prompt appears. If *practice* had already existed before this editing session, the changes you made are disregarded, but the file still exists.

This completes the tutorial. You learned how to get in and out of **vi**, insert and delete text, move the cursor around, make searches and replacements, how to execute line-oriented commands, copy text from other files, and cancel an editing session.

There are many more commands to learn, but this section covers the fundamentals of using **vi**. The following sections gives you more detailed information about these commands and about other **vi** commands and features.

Editing tasks

The following sections explain how to perform common editing tasks. By following the instructions in each section, you can complete each task described. Features that are needed in several tasks are described each time they are used, so some information is repeated.

How to enter the editor

There are several ways to begin editing, depending on what you are planning to do. This section describes how to start, or "invoke" the editor with one filename. To invoke **vi** on a series of files, see the section "Editing a series of files."

Editing a file

The most common way to enter **vi** is to enter the command **vi** and the name of the file you wish to edit:

vi *filename*

If ***filename*** does not already exist, this command creates a new, empty file.

Entering vi at a particular line

You can also enter the editor at a particular place in a file. For example, if you want to start editing a file at line 100, enter:

vi +100 *filename*

The cursor is placed at line 100 of ***filename.***

Entering vi at a particular word

If you wish to begin editing at the first occurrence of a particular word, enter:

vi+/*word filename*

The cursor is placed at the first occurrence of *word*. For example, to begin editing the file *temp* at the the first occurrence of *contain*, enter:

vi +/contain temp

Moving the cursor

The cursor movement keys allow you to move the cursor around in a file. Cursor movement can only be done in Command mode.

Moving the cursor by characters

The ⟨Space⟩ bar and the **l** key move the cursor forward a specified number of characters. The ⟨Bksp⟩ key and the **h** key move it backward a specified number of characters. If no number is specified, the cursor moves one character. For example, to move backward four characters, enter:

4h

You can also move the cursor to a designated character on the current line. **F** moves the cursor back to the specified character, **f** moves it forward. The cursor rests on the specified character. For example, to move the cursor backward to the nearest *p* on the current line, enter:

Fp

To move the cursor forward to the nearest *p*, enter:

fp

The **t** and **T** keys work the same way as **f** and **F**, but place the cursor immediately before the specified character. For example, to move the cursor back to the space next to the nearest *p* in the current line, enter:

Tp

If the *p* is in the word *telephone,* this command places the cursor on the *h*.

The cursor always remains on the same line when you use these commands. If you specify a number greater than the number of characters on the line, the cursor does not move beyond the beginning or end of that line.

Moving the cursor by words

The **w** key moves the cursor forward to the beginning of the specified number of words. Punctuation and non-alphabetic characters (such as ,!@#$%^&*()_+{}[]~| \'<>/) are considered words, so if a word is followed by a comma, the cursor counts the comma in the specified number.

For example, your cursor rests on the first letter of this sentence:

```
No, I didn't know he had returned.
```

If you press:

6w

the cursor stops on the *k* in *know*.

W works the same way as **w**, but includes punctuation and nonalphabetic characters as part of the word. Using the above example, if you press:

6W

the cursor stops on the *r* in *returned;* the comma and the apostrophe are included in their adjacent words.

The **e** and **E** keys move the cursor forward to the end of a specified number of words. The cursor is placed on the last letter of the word. The **e** command counts punctuation and nonalphabetic characters as separate words; **E** does not.

B and **b** move the cursor back to the beginning of a specified number of words. The cursor is placed on the first letter of the word. The **b** command counts punctuation and nonalphabetic characters as separate words; **B** does not. Using the example above, if the cursor is on the *r* in *returned,* enter:

4b

and the cursor moves to the *t* in *didn't.*

Enter:

4B

and the cursor moves to the first *d* in *didn't.*

The **w**, **W**, **b**, and **B** commands move the cursor to the next line if that is where the designated word is, unless the current line ends in a space.

Moving the cursor by lines

The ⟨Return⟩, ⟨Linefeed⟩ and ⟨+⟩ keys move the cursor forward a specified number of lines, placing the cursor on the first character. For example, to move the cursor forward six lines, enter:

6+

The **j** and ⟨Ctrl⟩n keys move the cursor forward a specified number of lines. The cursor remains in the same place on the line, unless there is no character in that place, in which case it moves to the last character on the line. For example, in the following two lines if the cursor is resting on the *e* in *characters,* pressing **j** moves it to the period at the end of the second line:

```
Lines contain characters.
Text contains lines█
```

The dollar sign($) moves the cursor to the end of a specified number of lines. For example, to move the cursor to the last character of the line four lines down from the current line, enter:

4$

⟨Ctrl⟩p and **k** move the cursor backward a specified number of lines, keeping it on the same place on the line. For example, to move the cursor backward four lines from the current line, enter:

4k

Moving the cursor on the screen

The **H**, **M**, and **L** keys move the cursor to the beginning of the top, middle and bottom lines of the screen, respectively.

Moving around in a file

The following commands move the file so different parts can be displayed on the screen. The cursor is placed on the first letter of the last line scrolled. Use ⟨Ctrl⟩**u** to scroll up one-half screen; ⟨Ctrl⟩**b** scrolls up a full screen. Use ⟨Ctrl⟩**d** to scroll down one-half screen; ⟨Ctrl⟩**f** scrolls down a full screen.

Placing a line at the top of the screen

To scroll the current line to the top of the screen, press **z**, then press ⟨Return⟩. To place a specific line at the top of the screen, precede **z** with the line number. For example:

33z

Press ⟨Return⟩, and line 33 scrolls to the top of the screen. For information on how to display line numbers, see the section "Displaying line numbers."

Inserting text before the cursor

You can begin inserting text before the cursor anywhere on a line, or at the beginning of a line. In order to insert text into a file, you must be in Insert mode. To enter Insert mode, press **i**. The "i" does not appear on the screen. Any text typed after the "i" becomes part of the file you are editing. To leave Insert mode and re-enter Command mode, press ⟨Esc⟩. For more explanation of modes in **vi**, see the section "Inserting text."

Use the uppercase "I" to enter Insert mode moves the cursor to the beginning of the current line. Use this to start inserting text at the beginning of the current line.

To append text after the cursor, press **a** to enter Insert mode (the "a" does not appear on your screen), then begin entering your text. Press ⟨Esc⟩ to leave Insert mode and re-enter Command mode.

If you use an uppercase "A" to enter Insert mode, this also moves the cursor to the end of the current line. It is useful for appending text at the end of the current line.

Correcting typing mistakes

If you make a mistake while you are typing, the simplest way to correct it is with the ⟨Bksp⟩ key. Backspace across the line until you have backspaced over the mistake, then retype the line. However, you can only do this if the cursor is on the same line as the error. See the sections "Deleting a character" through "Deleting an entire insertion" for other ways to correct typing mistakes.

Opening a new line

To open a new line above the cursor, press **O**. To open a new line below the cursor, press **o**. Both commands place you in Insert mode, and you can start entering text immediately. Press ⟨Esc⟩ to leave Insert mode and re-enter Command mode.

You may also use the ⟨Return⟩ key to open new lines above and below the cursor. To open a line above the cursor, move the cursor to the beginning of the line, press **i** to enter Insert mode, then press ⟨Return⟩. (For information on how to move the cursor, see the section "Moving the cursor.") To open a line below the cursor, move the cursor to the end of the current line, press **i** to enter Insert mode, then press ⟨Return⟩.

Repeating the last insertion

Pressing ⟨Ctrl⟩@ repeats the last insertion. Press **i** to enter Insert mode, then press ⟨Ctrl⟩@.

⟨Ctrl⟩@ only repeats insertions of 128 characters or less. If you inserted more than 128 characters, ⟨Ctrl⟩@ does nothing.

For other methods of repeating an insertion, see the sections "Inserting text from other files," and "Repeating a command."

Inserting text from other files

To insert the contents of another file into the file you are currently editing, use the Read (**r**) command. Move the cursor to the line immediately *above* the place you want the new material to appear, then enter:

:r *filename*

Where ***filename*** is the name of the file containing the material that you want to insert. The text of ***filename*** appears on the line below the cursor, and the cursor moves to the first character of the new text. This text is a copy; the original ***filename*** still exists.

Inserting selected lines from another file is more complicated. First, you copy the selected lines from the original file into a temporary holding place called a "buffer", then insert the text into the new file. Use these steps:

1. To select the lines to be copied, save your original file with the Write (**:w**) command , but do not exit **vi**.
2. Now, enter the following:

   :e *filename*

   The ***filename*** is the file that contains the text you want to copy.
3. Move the cursor to the first line you want to select.
4. Now, enter this command:

   mk

   This "marks" the first line of text to copy into the new file with the letter "k."
5. Move the cursor to the last line of the selected text and enter:

   "ay'k

   The lines from your first "mark" to the cursor are placed, or "yanked" into buffer *a*. The lines remain in buffer *a* until you replace them with other lines, or until you exit the editor.
6. Now, enter the following command to return to your previous file:

   :e#

   (For more information about this command, see the section "Editing a new File without leaving the editor.")
7. Move the cursor to the line above the place you want the new text to appear and enter:

   "ap

   This "puts" a copy of the yanked lines into the file, and the cursor is placed on the first letter of this new text. The buffer still contains the original yanked lines.

You can have 26 buffers named *a, b, c,* up to and including *z*. To name and select different buffers, replace the *a* in the above examples with whatever letter you want.

You can also delete text into a buffer, then insert it in another place. For information on this type of deletion and insertion, see the section "Moving text."

Copying lines from elsewhere in the file

To copy lines from one place in a file to another place in the same file, use the Copy (**co**) command.

co is a line-oriented command; to use it, you must know the line numbers of the text to be copied and its destination. To find out the number of the current line enter:

:nu

The line number and the text of that line are displayed on the status line. To find out the destination line number, move the cursor to the line above where you want the copied text to appear and repeat the **:nu** command. You can also make line numbers appear throughout the file with the **linenumber** option. For information on how to set this option, see the section "Displaying line numbers." The following example uses the **number** option to display line numbers in a file:

```
1 Files contain text.
2 Text contains lines.
3 Lines contain characters.
4 Characters form words.
5 Words form text.
~
~
~
~
~
```

Using the above example, to copy lines 3 and 4 and put them between lines 1 and 2, enter:

:3,4 co 1

The result is:

```
1 Files contain text.
2 Lines contain characters.
3 Characters form words.
4 Text contains lines.
5 Lines contain characters.
6 Characters form words.
7 Words form text.
~
~
~
```

If you have text that you want to insert several times in different places, you can save it in a buffer and insert it whenever you need it. For example, to repeat the first line of the following text after the last line:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

1. Move the cursor over the "F" in "Files". Enter the following command:

   "ayy

   This "yanks" the first line into buffer *a*.

2. Now, move the cursor over the "W" in "Words" and enter the following line:

   "ap

   This "puts" a copy of the yanked line into the file, and places the cursor on the first letter of this new text. The buffer still contains the original yanked line.

Your screen now looks like this:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
Files contain text.
~
~
~
~
```

If you want to yank several consecutive lines, indicate the number of lines you wish to yank after the name of the buffer. For example, to place three lines from the above text in buffer *a*, enter:

"a3yy

You can also use yank to copy parts of a line. For example, to copy the words "Files contain," enter:

```
2yw
```

This yanks the next two words, including the word on which you place the cursor. To yank the next ten characters, enter:

```
10yl
```

l indicates cursor motion to the right. To yank to the end of the line you are on, from where you are now, enter:

```
y$
```

Inserting control characters into text

Many control characters have special meaning in **vi**, even when you enter them in Insert mode. To remove their special significance, press ⟨Ctrl⟩**v** before typing the control character. Note that you cannot insert ⟨Ctrl⟩**j**, ⟨Ctrl⟩**q**, and ⟨Ctrl⟩**s** as text. ⟨Ctrl⟩**j** is a newline character; ⟨Ctrl⟩**q** and ⟨Ctrl⟩**s** are meaningful to the operating system, and are trapped by it before they are interpreted by **vi**.

Joining and breaking lines

To join two lines, press **J** while the cursor is on the first of the two lines you wish to join.

To break one line into two lines, position the cursor on the space preceding the first letter of what you want to be the second line and press **r** followed by ⟨Return⟩.

Deleting a character

Use the **x** and **X** commands to delete a specified number of characters. The **x** deletes the character above the cursor; **X** deletes the character immediately before the cursor. If no number is given, one character is deleted. For example, to delete three characters following the cursor (including the character above the cursor), enter:

```
3x
```

To delete three characters preceding the cursor, enter:

```
3X
```

Deleting a word

The **dw** command deletes a specified number of words. If no number is given, one word is deleted. A word is interpreted as numbers and letters separated by whitespace. When a word is deleted, the space after it is also deleted. For example, to delete three words, enter:

```
3dw
```

Deleting a line

The **D** command deletes all text following the cursor on that line, including the character the cursor is resting on. The **dd** command deletes a specified number of lines and closes up the space. If no number is given, only the current line is deleted. For example, to delete three lines, enter:

3dd

Another way to delete several lines is to use a line-oriented command. To use this command, it helps to know the line numbers of the text you want to delete. For information on how to display line numbers, see the section "Displaying line numbers."

For example, to delete lines 200 through 250, enter:

:200,250d

The following message appears on the **vi** status line, indicating how many lines were deleted:

```
50 lines
```

It is possible to remove lines without displaying line numbers using shorthand "addresses." For example, to remove all lines from the current line (the line the cursor rests on) to the end of the file, enter:

:.,$d

The dot (.) represents the current line, and the dollar sign stands for the last line in the file. To delete the current line and 3 lines following it, enter:

:.,+3d

To delete the current line and 3 lines preceding it, enter:

:.,-3d

For more information on using addresses in line-oriented commands, see **vi**(C) in the *User's Reference.*

Deleting an entire insertion

If you wish to delete all of the text you just entered, press ⟨Ctrl⟩**u** in Insert mode. The cursor returns to the beginning of the insertion. The text of the original insertion is still displayed, and any text you enter replaces it. When you press ⟨Esc⟩, any text remaining from the original insertion disappears.

Deleting and replacing text

Several **vi** commands combine removing characters and entering Insert mode. The following sections explain how to use these commands.

Overwriting characters

The **r** command replaces the character under the cursor with the next character entered. For example, to replace the character under the cursor with a "b", enter:

rb

If you give a number before **r**, that number of characters is replaced with the next character entered. For example, to replace the character above the cursor, plus the next three characters, with the letter "b", enter:

4rb

Note that you now have four "b"s in a row.

The **R** command replaces as many characters as you enter. To end the replacement, press ⟨Esc⟩. For example, to replace the second line in the following text with "Spelling is important.":

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Move the cursor over the "T" in "Text". Press **R**, then enter:

Spelling is important.

Press ⟨Esc⟩ to end the replacement. If you make a mistake, use the ⟨Bksp⟩ key to correct it. Your screen should now look like this:

```
Files contain text.
Spelling is important.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Substituting characters

The **s** command replaces a specified number of characters, beginning with the character under the cursor, with text you enter. For example, to substitute "xyz" for the cursor and two characters following it, enter:

3sxyz

The **S** command deletes a specified number of lines and replaces them with text you enter. You can enter as many new lines of text as you want; **S** affects only how many lines are deleted. If no number is given, one line is deleted. For example, to delete four lines, including the current line, enter:

4S

This differs from the **R** command. The **S** command deletes the entire current line; the **R** command deletes text from the cursor onward.

Replacing a word

The **cw** command replaces a word with text you enter. For example, to replace the word "bear" with the word "fox," move the cursor over the "b" in "bear" and enter:

cw

A dollar sign appears over the "r" in "bear", marking the end of the text that is being replaced. Now, enter:

fox

The rest of "bear" disappears and only "fox" remains.

Replacing the rest of a line

The **C** command replaces text from the cursor to the end of the line. For example, to replace the text of the sentence:

```
Who's afraid of the big bad wolf?
```

from "big" to the end, move the cursor over the "b" in "big" and enter **C**. A dollar sign (**$**) replaces the question mark (**?**) at the end of the line. Now, enter the following:

little lamb? ⟨Esc⟩

The remaining text from the original sentence disappears.

Replacing a whole line

The **cc** command deletes a specified number of lines, regardless of the location of the cursor, and replaces them with text you enter. If no number is given, the current line is deleted.

Replacing a particular word on a line

If a word occurs several times on one line, it is often convenient to use a line-oriented command to replace it. For example, to replace the word "removing" with "deleting" in the following sentence:

In vi, removing a line is as easy as removing a letter.

Make sure the cursor is at the beginning of that line, and enter:

:s/removing/deleting/g

This line-oriented command means "Substitute (s) for the word "removing" the word "deleting", everywhere it occurs on the current line (g)." If you don't include a "g" at the end, only the first occurrence of "removing" is changed.

For more information on using line-oriented commands to replace text, see the section "Searching and replacing."

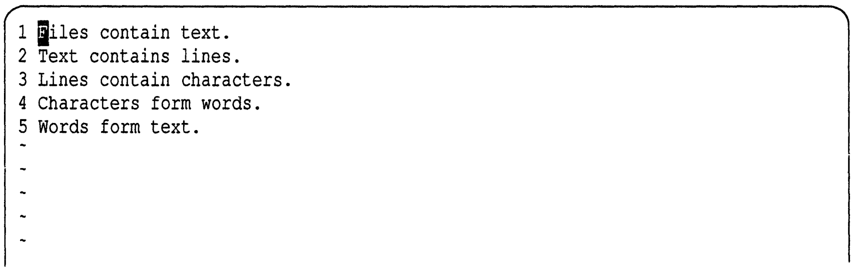
Moving text

To move a block of text from one place in a file to another, you can use the line-oriented **m** command. You must know the line numbers of your file to use this command. The **number** option displays line numbers. To set this option, press ⟨Esc⟩ to make sure you are in Command mode, then enter:

set number

Line numbers appear to the left of your text. For more information on setting the **number** option, see the section "Displaying line numbers." For other ways to display line numbers, see the section "Finding out what line you are on."

The following example uses the **number** option.



```
1 Files contain text.
2 Text contains lines.
3 Lines contain characters.
4 Characters form words.
5 Words form text.
~
~
~
~
~
```

To insert lines 2 and 3 between lines 4 and 5, enter:

:2,3m4

Your screen now looks like this:

```
1 Files contain text.
2 Characters form words.
3 Text contains lines.
4 Lines contain characters.
5 Words form text.
~
~
~
~
~
```

To place line 5 after line 2, enter:

:5m2

After moving, the screen looks like this:

```
1 Files contain text.
2 Characters form words.
3 Words form text.
4 Text contains lines.
5 Lines contain characters.
~
~
~
~
~
```

To make line 4 the first line in the file, enter:

:4m0

Your screen looks like this:

```
1 Text contains lines.
2 Files contain text.
3 Characters form words.
4 Words form text.
5 Lines contain characters.
~
~
~
~
~
```

You can also delete text into a buffer and insert it wherever you want. When text is deleted it is placed in a "delete buffer." There are nine delete buffers.

The first buffer always contains the most recent deletion. In other words, the first deletion in a given editing session goes into buffer 1. The second deletion also goes into buffer 1, and pushes the contents of the old buffer 1 into buffer 2. The third deletion goes into buffer 1, pushing the contents of buffer 2 into buffer 3, and the contents of buffer 1 into buffer 2. When buffer 9 has been used, the next deletion pushes the current text of buffer 9 off the stack and it disappears.

Text remains in the delete buffers until it is pushed off the stack, or until you quit the editor, so it is possible to delete text from one file, change files without leaving the editor, and place the deleted text in another file.

Delete buffers are particularly useful when you wish to remove text, store it, and put it somewhere else. Using the following text as an example:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Delete the first line by entering:

dd

Delete the third line the same way. Now move the cursor to the last line in the example and press:

"1p

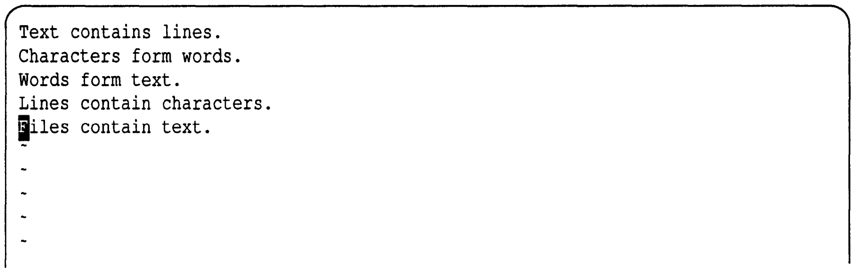
The line from the *second* deletion appears:

```
Text contains lines.
Characters form words.
Words form text.
Lines contain characters.
~
~
~
~
~
~
```

Now enter:

"2p

The line from the *first* deletion appears:



```
Text contains lines.
Characters form words.
Words form text.
Lines contain characters.
Files contain text.
~
~
~
~
~
```

Inserting text from a delete buffer does not remove the text from the buffer. Because the text remains in a buffer until it is either pushed off the stack or until you quit the editor, you can use it as many times as you want.

You can also place text in named buffers. For information on how to create named buffers, see the section "Inserting text from other files."

Searching for patterns

You can search forward and backward for patterns in **vi**. To search forward, press the slash (/) key. The slash appears on the status line. Enter the characters you want to search for and press ⟨Return⟩. If the specified pattern exists, the cursor moves to the first character of the pattern.

For example, to search forward in the file for the word "account," enter:

/account

The cursor moves to the first character of the pattern. To place the cursor at the beginning of the line above "account," for example, enter:

/account/-

To place the cursor at the beginning of the line two lines above the line that contains "account," enter:

/account/-2

To place the cursor two lines below "account," enter:

/account/+2

To search backward through a file, use " ? " instead of " / " to start the search. For example, to find all occurrences of "account" above the cursor, enter:

?account

To search for a pattern containing any of the special characters ". * \ [ ] ~ $" and "^", each special character must be preceded by a backslash. For example, to find the pattern "U.S.A.", enter:

```
/U\.S\.A\./
```

You can continue to search for a pattern by pressing **n** after each search. The pattern is unaffected by intervening **vi** commands, and you can use **n** to search for the pattern until you enter a new pattern or quit the editor.

vi searches for exactly what you enter. If the pattern you are searching for contains an uppercase letter (for example, if it appears at the beginning of a sentence), **vi** ignores it. To disregard case in a search command, you can set the **ignorecase** option:

```
:set ignorecase
```

By default, searches "wrap around" the file. That is, if a search starts in the middle of a file, when **vi** reaches the end of the file, it "wraps around" to the beginning, and continues until it returns to where the search began. Searches complete faster if you specify forward or backward searches, depending on where you think the pattern is.

If you do not want searches to wrap around the file, you can change the **wrapscan** option. Enter:

```
:set nowrapscan
```

For more information about setting options, see the section "Setting up your vi environment."

Searching and replacing

The search and replace commands allow you to perform complex changes to a file in a single command. If you are a serious user of **vi**, you must learn how to use these commands.

The syntax of a search and replace command is:

```
g/pattern1/s/[pattern2]/[options]
```

Brackets indicate optional parts of the command line. The **g** tells the computer to execute the replacement on every line in the file; otherwise, the replacement occurs only on the current line. The *options* are explained in the following sections.

To explain these commands, we will use the example file from the tutorial section:

```
Files contain text.
Text contains lines.
Lines contain characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Replacing a word

To replace the word "contain" with the word "are" throughout the file, enter the following command:

> **:g/contain /s//are /g**

This command says "On each line of the file (g), find "contain" and substitute for that word (s//) the word "are", everywhere it occurs on that line (the second g)." Note that a space is included in the search pattern for "contain"; without the space "contains" is also replace.

After the command executes, your screen looks like this:

```
Files are text.
Text contains lines.
Lines are characters.
Characters form words.
Words form text.
~
~
~
~
~
```

Printing all replacements

To replace "contain" with "are" throughout the file, and print every line changed, use the **p** option:

> **:g/contain /s//are /gp**

After the command executes, each line in which "contain" was replaced by "are" is printed on the lower part of the screen. To remove these lines, redraw the screen by pressing ⟨Ctrl⟩r.

Choosing a replacement

Sometimes you might not want to replace every instance of a given pattern. The **c** option displays every occurrence of ***pattern*** and waits for you to confirm that you want to make the substitution. If you press **y** the substitution takes place; if you press ⟨Return⟩ the next instance of ***pattern*** is displayed.

To run this command on the example file, enter:

```
:g/contain/s//are/gc
```

The first instance of "contain" appears on the status line:

```
Files contain text.
      ^^^^^^^
```

Press **y** , then ⟨Return⟩. The next occurrence of "contain" appears.

Pattern matching

In addition to the characters you want to find, search commands often require a context in which you want to find them. For example, you might want to locate every occurrence of a word at the beginning of a line. **vi** provides several special characters that specify particular contexts.

Matching the beginning of a line

When you place a caret (^) at the beginning of a pattern, only patterns found at the beginning of a line are matched. For example, the following search pattern only finds "text" when it occurs as the first word on a line:

```
/^text/
```

To search for a caret that appears as text, you must precede it with a backslash (\).

Matching the end of a line

When you place a dollar sign ($) at the end of a pattern, only patterns found at the end of a line are matched. For example, the following search pattern only finds "text" when it occurs as the last word on a line:

```
/text$/
```

To search for a dollar sign that appears as text you must precede it with a backslash (\).

Matching any single character

When used in a search pattern, the period (.) matches any single character except the newline character. For example, to find all words that end with "ed," use the following pattern:

```
/.ed /
```

Note the space between the *d* and the backslash.

To search for a period in the text, you must precede it with a backslash (\).

Matching a range of characters

A set of characters enclosed in square brackets matches any single character in the range designated. For example, to find any lowercase letter, enter the following search pattern:

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

To find all occurrences of "apple" and "Apple", use this search pattern:

```
/[aA]pple/
```

To search for a bracket that appears as text, you must precede it with a backslash (\).

Matching exceptions

A caret (^) at the beginning of ***string*** matches every character *except* those specified in ***string***. For example, the following search pattern finds anything but a lowercase letter or a newline:

```
[^a-z]
```

Matching the special characters

To place a caret, hyphen or square bracket in a search pattern, precede it with a backslash. For example, to search for a caret, enter:

```
/\^/
```

If you need to search for many patterns that contain special characters, you can reset the **magic** option. To do this, enter:

```
:set nomagic
```

This removes the special meaning from the ., \, $, [and] characters. You can include them in search and replace commands without a preceding backslash. Note that you cannot remove the special meaning removed from the special characters star (*) and caret (^); these must always be preceded by a backslash in searches.

To restore **magic**, enter:

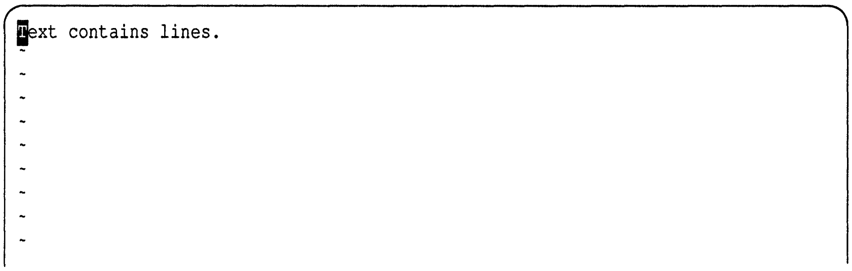
```
:set magic
```

For more information about setting options, see the "Setting up your vi environment" section.

Undoing a command

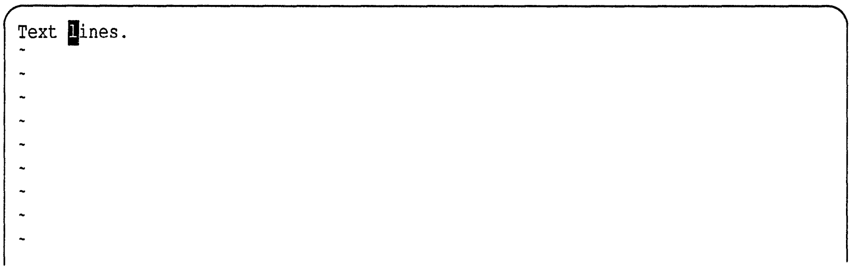
You can reverse any editing command with the Undo (**u**) command. The Undo command works on both screen-oriented and line-oriented commands. For example, if you delete a line and then decide to keep it, press **u** and the line reappears.

Use the following line as an example:



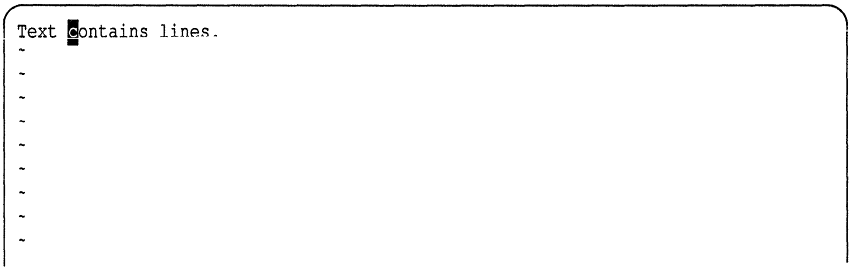
```
Text contains lines.
~
~
~
~
~
~
~
~
~
```

Place the cursor over the "c" in "contains," then delete the word with the **dw** command. Your screen now looks like this:



```
Text lines.
~
~
~
~
~
~
~
~
~
```

Press **u** to undo the **dw** command; "contains" reappears:



```
Text contains lines.
~
~
~
~
~
~
~
~
~
```

If you press **u** again, "contains" is deleted again:

```
Text lines.
~
~
~
~
~
~
~
~
~
```

It is important to remember that **u** only undoes the *last* command. For example, if you make a global search and replace, then delete a few characters with the **x** command, pressing **u** undoes the deletions but not the global search and replace.

Repeating a command

You can repeat any screen-oriented **vi** command with the Repeat (**.**) command. For example, you delete two words by entering:

2dw

You can repeat this command as many times as you wish by pressing the period key (**.**). Cursor movement does not affect the Repeat command, so you can repeat a command as many times and in as many places in a file as you want.

The Repeat command only repeats the last **vi** command. Careful planning can save time and effort. For example, if you want to replace a word that occurs several times in a file (and for some reason you do not wish to use a global command), use the **cw** command instead of deleting the word with the **dw** command, then inserting new text with the **i** command. By using the **cw** command you can repeat the replacement with the dot (**.**) command. If you delete the word, then insert new text, dot only repeats the replacement.

Leaving the editor

There are several ways to exit the editor and save any changes you may have made to the file. One way is to enter **:x**. This command replaces the old copy of the file with the new one you have just edited, quits the editor, and returns you to the UNIX shell. Similarly, if you enter **ZZ** the same thing happens, except the old copy file is replaced *only* if you have made any changes. Note that the **ZZ** command is *not* preceded by a colon, and is not echoed on the screen.

To leave the editor without saving any changes you have made to the file, enter:

```
:q!
```

The exclamation point tells **vi** to quit unconditionally. If you leave out the exclamation point, **vi** does not let you quit:

```
:q
```

You see the following error message:

```
No write since last change (:quit! overrides)
```

This message tells you to use **:q!** if you really want to leave the editor without saving your file.

Saving a file without leaving the editor

There are many occasions when you must save a file without leaving the editor, such as when starting a new shell, or moving to another file. Before you can perform these tasks, you must first save the current file with the Write (**:w**) command:

```
:w
```

You do not need to enter the name of the file; **vi** remembers the name you used when you invoked the editor. If you invoked **vi** without a filename, you can name the file by entering:

:w ***filename***

The ***filename*** is the name of the new file.

Editing a series of files

Entering and leaving **vi** for each new file takes time, particularly on a heavily used system, or when you are editing large files. If you have many files to edit in one session, you can invoke **vi** with more than one filename, and thus edit more than one file without leaving the editor, as in:

```
vi file1 file2 file3 file4 file5 file6
```

But entering many filenames is tedious, and you may make a mistake. If you mistype a filename, you must either backspace over the mistake and re-enter the line, or kill the whole line and re-enter it. It is more convenient to invoke **vi** using the special characters as abbreviations.

To invoke **vi** on the above files without typing each name, enter:

```
vi file*
```

This invokes **vi** on all files that begin with the letters "file." You can plan your filenames to save time in later editing. For example, if you are writing a document that consists of many files, it would be wise to give each file the same filename extension, such as ".s." Then you can invoke **vi** on the entire document:

```
vi *.s
```

You can also invoke **vi** on a selected range of files:

```
vi [3-5]*.s
```

or

```
vi [a-h]*
```

To invoke **vi** on all files that are five letters long, and have any extension:

```
vi ?????.*
```

For more information on using special characters, see the *Tutorial.*

When you invoke **vi** with more than one filename, you see the following message when the first file is displayed on the screen:

```
*x files to edit
```

After you have finished editing a file, save it with the Write (**:w**) command, then go to the next file with the Next (**:n**) command:

```
:n
```

The next file appears, ready to edit. It is not necessary to specify a filename; the files are invoked in alphabetical (or numerical, if the filenames begin with numbers) order.

If you forget what files you are editing, enter:

```
:args
```

The list of files appears on the status line. The current file is enclosed in square brackets.

To edit a file out of order, such as *file4* after *file2*, enter the following command instead of using the (**:n**) command:

```
:e file4
```

If you enter **:n** after you finish editing *file4*, you return to *file3*.

If you wish to start again from the beginning of the list, enter:

```
:rew
```

To discard the changes you made and start again at the beginning, enter:

```
:rew!
```

Editing a new file without leaving the editor

You can start editing another file anywhere on a UNIX system without leaving **vi**. This saves time when you want to edit several files in one session that are in different directories, or even in the same directory. For example, if you have finished editing */usr/joe/memo* and you wish to edit */usr/mary/letter*. First, save the file *memo* with the Write (**:w**) command. Then, enter:

:e /usr/mary/letter

/usr/mary/letter appears on your screen just as though you had left **vi**.

NOTE You *must* write out your file with the Write (**:w**) command to save the changes you have made. If you try to edit a second file without writing out the first file, the following message appears:

```
No write since last change (:e! overrides)
```

If you use **:e!** all your changes to the first file are discarded.

If you want to switch back and forth between two files, **vi** remembers the name of the last file edited. Using the above example, if you wish to go back and edit the file */usr/joe/memo* after you have finished with */usr/mary/letter*, enter:

:e#

The cursor is positioned in the same location it was when you first saved */usr/joe/memo*.

Leaving the editor temporarily

You can execute any UNIX command from within **vi** using the shell Escape (**!**) command. For example, if you wish to find out the date and time, enter:

:!date

The exclamation point sends the remainder of the line to the shell to be executed, and the date and time appear on the **vi** status line. You can use the **!** to perform any UNIX command. To send mail to joe without leaving the editor, enter:

:!mail joe

Type your message and send it. (For more information about the UNIX mail system, see the "Communicating using mail" chapter.) After you send the mail message, the following message appears:

```
[Press return to continue]
```

Press ⟨Return⟩ to continue editing.

If you want to perform several UNIX commands before returning to the editor, you can invoke a new shell:

:!sh

The UNIX prompt appears. You can execute as many commands as you like. Press ⟨Ctrl⟩d to terminate the new shell and return to your file.

If you have not written out your file before a shell escape, you see the following message:

```
[No write since last change]
```

It is a good idea to save your file with the Write (**:w**) command before executing an escape, just in case something goes wrong. However, once you become an experienced **vi** user, you may wish to turn off this message. To turn off the "No write" message, reset the **warn** option, as follows:

:set nowarn

For more information about setting options in **vi**, see the section "Setting up your vi environment."

Performing a series of line-oriented commands

If you have several line-oriented commands to perform, you can place yourself temporarily in line-oriented mode by entering **Q** while you are in Command mode. A colon prompt appears on the status line.

You cannot undo commands executed in this mode with the **u** command, nor do they appear on the screen until you re-enter Normal **vi** mode. To re-enter Normal **vi** mode, enter:

vi

Finding out what file you are editing

If you forget what file you are editing, press ⟨Ctrl⟩g in Command mode. A line similar to the following appears on the status line:

```
"memo" [Modified] line 12 of 100 --12%--
```

From left to right, the following information is displayed:

- the name of the file
- whether or not the file has been modified
- the line number the cursor is on
- how many lines there are in the file
- your location in the file (expressed as a percentage)

This command is also useful when you need to know the line number of the current line for a line-oriented command.

The same information can be obtained by entering:

```
:file
```

or

```
:f
```

Finding out what line you are on

To find out what line of the file you are on, enter:

```
:nu
```

This command displays the current line number and the text of the line.

To display line numbers for the entire file, see the section "Displaying line numbers."

Solving common problems

The following is a list of common problems that you might encounter when using **vi**, along with the probable solution:

- *I do not know which mode I am in.*

  Press ⟨Esc⟩ until the bell rings. When the bell rings you are in Command mode.

- *I cannot get out of a subshell.*

  Press ⟨Ctrl⟩d to exit any subshell. If you have created more than one subshell (not a good idea, usually), keep pressing ⟨Ctrl⟩d until you see the following message:

  ```
  [Press return to continue]
  ```

- *I made an inadvertent deletion (or insertion).*

  Press **u** to undo the last Delete or Insert command.

- *There are extra characters on my screen.*

  Press ⟨Ctrl⟩l to redraw the screen.

- *When I type, nothing happens.*

  vi has crashed and you are now in the shell with your terminal characteristics set incorrectly. To reset the keyboard, slowly enter:

  ```
  stty sane
  ```

then press ⟨Ctrl⟩**j** or ⟨Linefeed⟩. Pressing ⟨Ctrl⟩**j** instead of ⟨Return⟩ is important here, because it is quite possible that the ⟨Return⟩ key will not work as a newline character. To make sure that other terminal characteristics have not been altered, log off, turn your terminal off, turn your terminal back on, and then log back in. This should guarantee that your terminal's characteristics are back to normal. This procedure might vary somewhat depending on the terminal.

- *The system crashed while I was editing.*

  Normally, **vi** informs you (by sending you mail) that your file was saved before a crash. You can recover your file by entering:

  ```
  vi -r filename
  ```

  If **vi** was unable to save the file before the crash, it is irretrievably lost.

- *I keep getting a colon on the status line when I press* ⟨**Return**⟩.

  You are in line-oriented Command mode; enter the following command to return to normal **vi** Command mode:

  ```
  vi
  ```

- *I get the error message "Unknown terminal type [Using open mode]" when I invoke vi.*

  Your terminal type is not set correctly. To leave Open mode, press ⟨Esc⟩, then enter:

  ```
  :wq
  ```

  Turn to the section "Setting the terminal type" for information on how to set your terminal type correctly.

Setting up your vi environment

There are a number of options that you can set that affect your terminal type, how **vi** displays files and error messages on your screen, and how searches are performed. You can set these options with the **set** command while you are editing, with the **EXINIT** environment variable (see the **environ**(M) manual page), or you can place them in the **vi** *.exrc* startup file (see "Customizing your environment").

You can also define mappings and abbreviations to reduce repetitive tasks with the **map** and **abbr** commands while you are editing, with **EXINIT**, or in the *.exrc* file.

The following sections describe how to set some commonly used options and how to create mappings and abbreviations. There is a complete list of options in **vi**(C) in the *User's Reference.*

Setting the terminal type

Before you can use **vi**, you must set the terminal type, if this has not already been done for you, by defining the **TERM** variable in your *.profile* or *.login* file. The **TERM** variable is a number that tells the operating system what type of terminal you are using. To determine this number, you must find out what type of terminal you are using. Then, look up this type in **terminals**(M) in the *User's Reference*. If you cannot find your terminal type or its number, consult your system administrator.

For these examples, we suppose that you are using an HP 2621 terminal. For the HP 2621, the **TERM** variable is "2621". How you define this variable depends on the shell that you are using. You can usually determine the shell by examining the prompt character. The Bourne and Korn shells prompt with a dollar sign ($); the C shell prompts with a percent sign (%).

If you are using the Bourne or Korn shells, set your terminal type to 2621 by placing the following commands in the *.profile* file:

```
TERM=2621
export TERM
```

For the C shell, set your terminal type to 2621 by placing the following command in the *.login* file:

```
setenv TERM 2621
```

Setting options with the set command

Use the **set** command while you are editing in **vi** to display the current option settings and to set options.

Listing the available options

To get a list of the options available to you and how they are currently set, enter:

```
:set all
```

Your display looks similar to this:

```
noautoindent     open                      noslowopen
autoprint        nooptimize                tabstop=8
noautowrite      paragraphs=IPLPPPQPP LIbp taglength=0
nobeautify       noprompt                  ttytype=h19
directory=/tmp   noreadonly                term=h19
noerrorbells     redraw                    noterse
hardtabs=8       report=5                  warn
noignorecase     scroll=4                  window=8
nolisp           sections=NHSHH HU         wrapscan
nolist           shell=/bin/sh             wrapmargin=0
magic            shiftwidth=8              nowriteany
nonumber         noshowmatch
```

This section discusses only the most commonly used options. For information about the options not covered here, see **vi**(C) in the *User's Reference.*

Setting an option

To set an option, use the **set** command. For example, to set the **ignorecase** option so that case is *not* ignored in searches, enter:

:set noignorecase

Displaying tabs and end-of-line

The **list** option tells **vi** to display the "hidden" characters and end-of-line. The default setting is **nolist**. To display these characters, enter:

:set list

This redraws your screen, displaying these characters. The dollar sign ($) represents end-of-line and ⟨Ctrl⟩i (^I) represents the ⟨Tab⟩ character.

Ignoring case in search commands

By default, case is significant in search commands. To disregard case in searches, enter:

```
:set ignorecase
```

To return to the default setting, enter:

```
:set noignorecase
```

Displaying line numbers

To display the line numbers in the file, enter:

```
:set number
```

This redraws your screen, placing numbers to the left of the text.

Printing the number of lines changed

The **report** option tells you the number of lines modified by a line-oriented command. For example, to tell **vi** to report the number of lines modified if more than one line is changed, enter this command:

```
:set report=1
```

The default setting is:

```
report=5
```

This reports the number of lines changed when more than five lines are modified.

Changing the terminal type

If you are logged in on a terminal that is a different type than the one you normally use, you can check the terminal type setting by entering:

```
:set term
```

See the section "Setting the terminal type" for more information about the **TERM** variable.

Shortening error messages

After you become experienced with **vi**, you might want to shorten your error messages. To change from the default **noterse**, enter:

```
:set terse
```

As an example of the effect of **terse**, when **terse** is set the following message:

```
No write since last change, quit! overrides
```

becomes:

```
No write
```

Turning off warnings

After you become experienced with **vi**, you might want to turn off the error message that appears if you have not written out your file before a Shell Escape (**:!**) command. To turn these messages off, enter:

:set nowarn

Permitting special characters in searches

The **nomagic** option allows the inclusion of the special characters " . \ $ [ ] " in search patterns without a preceding backslash. This option does *not* affect caret (^) or star (*); you must always precede these characters with a backslash in searches regardless of **magic**. To set **nomagic**, enter:

:set nomagic

Limiting searches

By default, searches in **vi** "wrap" around the file until they return to the place they started. To save time, you might want to disable this feature. Use the following command:

:set nowrapscan

When you set this option, forward searches go only to the end of the file, and backward searches stop at the beginning.

Turning on messages

If someone sends you a message with the **write**(C) command while you are in **vi**, the text of the message appears on your screen. To remove the message from your display, you must press ⟨Ctrl⟩**l**. When you invoke **vi**, write permission to your screen is automatically turned off, preventing **write** messages from appearing. If you want to receive **write** messages while in **vi**, reset this option as follows:

:set mesg

Mapping keys

Use the **map** command to map any character or escape sequence to a command sequence. For example, with the following command defined, when you enter the number sign (#) in Command mode, **vi** adds a semicolon to the end of the current line:

```
map # A;^[
```

⟨Ctrl⟩[represents the ⟨Esc⟩ key that you must enter to exit from Insert mode. When you create a mapping, use ⟨Ctrl⟩**v** to escape control characters.

Here is a more complex example:

```
map ^P :w^M:!spell %^M
```

⟨Ctrl⟩**p** key is mapped to two commands. First, it writes the file, then it executes a shell escape to run the spell checker on the current file (represented by the percent sign). The ⟨Ctrl⟩**m** represents the ⟨Return⟩ you must enter to execute each command.

Be careful not to map keys that are already defined within **vi**, such as ⟨Ctrl⟩**r**, which is defined by default to redraw the screen.

You can remove a mapping with the **unmap** command.

Abbreviating strings

The **abbr** command allows you to avoid typing a frequently used word or phrase by mapping a short string to a longer string. For example, with the following command defined, when you enter "Usa" in Insert mode, **vi** expands the string to "United States of America."

```
:abbr Usa United States of America
```

When you create an abbreviation, it helps to use mixed case (as in "Usa") so that you can still enter "USA" if you need to without it expanding.

You can remove an abbreviation with the **unabbreviate** command.

Customizing your environment

Each time you invoke **vi**, it reads commands from the file named *.exrc* in your home directory. This file sets your preferred options so that you do not need to set them each time you invoke **vi**. A sample *.exrc* file follows:

```
set number
set ignorecase
set nowarn
set report=1
map ^W !}fmt^M
abbr unix \s-1UNIX\s+1
```

Each time you invoke **vi** with the above settings, your file is displayed with line numbers, case is ignored in searches, warnings before shell escape commands are turned off, and any command that modifies more than one line displays a message indicating how many lines were changed. In addition, the ⟨Ctrl⟩**w** key is defined to escape to the shell to run a formatting command on the current paragraph, and the string "unix" is defined to expand to a string containing **troff**(CT) commands that print small capital letters.

Printing Files

To print files, use the **lp** command. This is one of a group of commands known as the "lineprinter" commands. The lineprinter commands are easy to use and very flexible. With a few simple commands, you can print multiple copies of a file, cancel a print request, or ask for a special option on a particular printer. Check with your system administrator to find out what lineprinters and printer options are available on your system.

Using lp

This section explains how to print the *temp* file that you created in the tutorial section.

A directory must be "publicly executable" before you can use **lp** to print any of the files in that directory. This means that other users must have execute permissions on the directory. Enter the following command to make your home directory publicly executable:

```
chmod o+x $HOME
```

(See Chapter 3, "Managing files and directories," for more information on **chmod**(C).)

Enter the following command to print *temp*:

```
lp temp
```

This command causes one copy of *temp* to print on the default printer on your system. A banner page might be printed along with the file. Note that you can print several files at once by putting more than one name on the **lp** command line.

When you print with **lp**, a "request ID" like the following is displayed on your screen:

```
pr4-532
```

The first part (pr4) is the name of the printer on which your file is printed. The second part (532) identifies your job number. If you want to cancel your print request or check its status later, you need to remember your request ID. (The following sections discuss canceling and checking on print requests.)

You can also use **lp** with pipes. (If you are not familiar with pipes, see the section "Commands revisited: pipes and redirection," in the *Tutorial.*) For example, enter the following command to sort and then print a copy of */etc/passwd,* the file that contains system account information:

```
sort /etc/passwd | lp
```

(For more information on **sort**(C), see the section "Sorting files," later in this chapter.)

Using lp options

The **lp** command has several options that help you control the printed output. You can specify the number of copies you want printed by using the number option, **-n**. For example, to print two copies of *temp,* enter:

```
lp -n2 temp
```

Several different printers are often attached to a single UNIX system. With the **-d** option, you can specify the printer on which your file is to be printed. To print two copies of *temp* on a printer named *laser,* enter:

```
lp -n2 -dlaser temp
```

Check with your system administrator for the names of the printers available on your system.

Canceling a print request

Use the **cancel** command to cancel any of your print requests. With this command, you can only cancel your own requests. The system administrator is the only person allowed to cancel the requests of other users. If the system administrator cancels one of your print requests, you are automatically notified via mail.

The **cancel** command takes as its argument the request ID. For example, to stop printing one of your files with a request ID of *laser-245,* you enter:

```
cancel laser-245
```

Experiment with **cancel** by using **lp** to print *temp* and then using **cancel** to cancel the print request.

The system administrator can also use the **cancel** command to stop whatever is currently printing on a particular printer. For example, to cancel whatever file is currently printing on the printer *laser*, enter the following command:

cancel laser

If you cancel a file that does not belong to you, mail reporting that the print request was canceled is automatically sent to the file's owner.

Finding out the status of a print request

Use the **lpstat** command to check on the status of your print request. To use it, simply enter the following:

lpstat

The **lpstat** command produces output like the following:

```
prtl-121    cindym    450   Dec 15 09:30
laser-450   cindym   4968   Dec 15 09:46
```

Note that entering **lpstat** with no options displays information on your files only, not those of other users. To generate a report for all users on your computer, use **lpstat** with the **-o** option. Nothing is displayed by the **lpstat** command if the print job is complete.

NOTE Depending on how your system administrator has set up your system, you may not be allowed to see other print jobs. Refer to Chapter 7, "Using a secure system," for details.

The first column of the **lpstat** output shows the request ID for each of your files being printed. The second column is your login name. The third column shows the number of characters to be printed and the fourth column lists the dates and times the print requests were made.

To learn the status of a particular file, use the **lpstat** command with the file's request ID. For example, to find out the status of a file with the request ID of *laser-256*, enter the following command:

lpstat laser-256

The status of that file only is displayed.

You can also request the status of various printers on your system by using the **-p** option or by giving the name of the particular printer you are interested in. Enter the following command to find out the status of all the printers on your system:

lpstat -p

To find out the status of a printer named *laser*, enter the following:

lpstat -plaser

The request ID and status information for each file currently waiting to be printed on *laser* is displayed.

Processing text files

The UNIX system includes a set of utilities that let you process information in text files. These utilities enable you to compare the contents of two files, sort files, search for patterns in files, and count the characters, words, and lines in files. These utilities are described below.

Comparing files

The **diff**(C) command allows you to compare the contents of two files and to print out those lines that differ between the files. To experiment with **diff**, use **vi**(C) to create two files called *men* and *women* to compare. Add the following lines to *men*:

```
Now is the time for all good men to
Come to the aid of their party.
```

And, add the following lines *women*:

```
Now is the time for all good women to
Come to the aid of their party.
```

Enter the following command to compare the contents of these two files:

diff men women

This **diff** command produces the following output:

```
1c1
< Now is the time for all good men to
---
> Now is the time for all good women to
```

The lines displayed are the lines that differ from one another in the two files.

Sorting files

One of the most useful file processing commands is **sort**(C). When you use **sort** without any options, this command alphabetizes lines in a file, starting with the leftmost character of each line. By default, **sort** outputs the sorted lines to the screen. You can use redirection on the **sort** line to place the output the **sort** command in a file. The **sort** command does not affect the contents of the input file.

Enter the following command to display an alphabetized list of all users who have system accounts:

```
sort /etc/passwd
```

The **sort** command is useful in pipes. For example, enter the following command to display an alphabetized list of users who are currently using the system:

```
who | sort
```

Removing repeated lines in a file

Use the **uniq**(C) command to remove adjacent lines that are repeated in the file. For example, if you sort a file that contains duplicate entries, you can use **uniq** to remove the redundant entries from the output. Thus, to sort and remove redundant entries from a list of phone numbers, use the following command:

```
sort phone.list | uniq
```

This outputs the lines in the *phone.list* file that are unique. Note that **uniq** only removes redundant entries if they are adjacent.

For more information on the options to **uniq**, see the **uniq**(C) manual page.

Searching for patterns in a file

Use the **grep**(C) command to select and extract lines from a file, and print only those lines that match a given pattern. Enter the following command to print out the lines in */etc/passwd* that contain your login information. Generally, there is only one such line:

```
grep login /etc/passwd
```

Be sure to replace ***login*** in this command with your login name. Your output should be similar to the following:

```
markt:+:6005:104:Mark Taub, Docland:/u/markt:/bin/csh
```

Note that whenever you use wildcard characters to specify a **grep** search pattern, you should enclose the pattern in single quotation marks (′). Note also that the search pattern is case sensitive. Searching for "joe" does not yield lines containing "Joe."

As another example, assume that you have a file named *phonelist* that contains a name followed by a phone number on each line. Assume also that there are several thousand lines in this list. You can use **grep** to find the phone number of someone named Joe, whose phone number prefix is 822, by entering the following command:

```
grep ′Joe′ phonelist | grep ′822-′ > joes.number
```

The **grep** utility first finds all occurrences of lines containing the word "Joe" in the file *phonelist*. The output from this command is then filtered through another **grep** command, which searches for an "822-" prefix, thus removing any unwanted "Joes." Finally, assuming that a unique phone number for Joe exists with the "822-" prefix, **grep** places that name and number in the file called *joes.number*.

Two other pattern searching utilities are available. These are **egrep**(CM) and **fgrep**(CM). Refer to the **grep**(C) manual page in the *User's Reference* for more information on these utilities.

Counting words, lines, and characters

Use the **wc**(C) utility to count words in a file. Words are defined as a character or group of characters separated by punctuation, spaces, tabs, or newlines. In addition to counting words, **wc** counts characters and lines.

For example, use **wc** to count the lines, words, and characters in the *men* file that you created earlier in the "Comparing files" section:

```
wc men
```

The output from this command should be the following:

```
2     16     68 men
```

The first number is the number of lines in *men*, the second number is the number of words and the third number is the number of characters.

To specify a count of characters, words, or lines only, use the **-c**, **-w**, or **-l** option, respectively. For example, enter the following command to count the number of users currently logged onto the system:

```
who | wc -l
```

The **who** command reports on who is using the system, one user per line. The **wc -l** command counts the number of lines reported by the **who** command. This is the number of users currently on the system.

Summary

In addition to the basic **vi** commands and variables, this chapter covers the commands and utilities that you can use to print and process text files. This section contains a summary of the commands discussed in this chapter.

Summary of vi commands and variables

The following tables contain all the basic **vi** commands and variables discussed in this chapter.

Table 2-1 Entering vi

Typing this:	Does this:
vi ***file***	Starts at line 1
vi +n ***file***	Starts at line *n*
vi + ***file***	Starts at last line
vi +/*pattern* *file*	Starts at *pattern*
vi -r ***file***	Recovers *file* after a system crash

Table 2-2 Cursor movement

Pressing this key:	Does this:
h	Moves 1 space left
l	Moves 1 space right
⟨Space⟩	Moves 1 space right
w	Moves 1 word right
b	Moves 1 word left
k	Moves 1 line up
j	Moves 1 line down
⟨Return⟩	Moves 1 line down
)	Moves to end of sentence
(	Moves to beginning of sentence
}	Moves to beginning of paragraph
{	Moves to end of paragraph
⟨Ctrl⟩w	Moves to first character of insertion
⟨Ctrl⟩u	Scrolls up 1/2 screen
⟨Ctrl⟩d	Scrolls down 1/2 screen
⟨Ctrl⟩f	Scrolls down one screen
⟨Ctrl⟩b	Scrolls up one screen

Table 2-3 Inserting text

Pressing this key:	Starts insertion:
i	Before the cursor
I	Before first character on the line
a	After the cursor
A	After last character on the line
o	On next line down
O	On the line above
r	On current character, replaces one character only
R	On current character, replaces until ⟨Esc⟩

Table 2-4 Delete commands

Command	Function
dw	Deletes a word
d0	Deletes to beginning of line
d$	Deletes to end of line
3dw	Deletes 3 words
dd	Deletes the current line
5dd	Deletes 5 lines
x	Deletes a character

Table 2-5 Change commands

Command	Function
cw	Changes 1 word
3cw	Changes 3 words
cc	Changes current line
5cc	Changes 5 lines

Table 2-6 Search commands

Command	Function	Example
***/and**	Finds the next occurrence of *and*	and, stand, grand
***?and**	Finds the previous occurrence of *and*	and, stand, grand
***/^The**	Finds next line that starts with *The*	The, Then, There
/[bB]ox/	Finds the next occurrence of *box* or *Box*	
n	Repeats the most recent search, in the same direction	

Table 2-7 Search and replace commands

Command	Result	Example
:s/pear/peach/g	All *pear*s become *peach* on the current line	
:1,$s/file/directory	Replaces *file* with *directory* from line 1 to the end.	*filename* becomes *directoryname*
:g/one/s//1/g	Replaces every occurrence of *one* with 1.	one becomes 1, oneself becomes 1self, someone becomes some1

Table 2-8 Pattern matching: special characters

This character:	Matches:
^	Beginning of a line
$	End of a line
.	Any single character
[]	A range of characters

Table 2-9 Leaving vi

Command	Result
:w	Writes out the file
:x	Writes out the file, quits **vi**
:q!	Quits **vi** without saving changes
:!command	Executes *command*
:!sh	Forks a new shell
!!command	Executes *command* and places output on current line
:e *file*	Edits *file* (save current file with **:w** first)

Table 2-10 Options

This option:	Does this:
all	Lists all options
term	Sets terminal type
ignorecase	Ignores case in searches
list	Displays ⟨Tab⟩ and end-of-line characters
number	Displays line numbers
report	Prints number of lines changed by a line-oriented command
terse	Shortens error messages
warn	Turns off "no write" warning before escape
nomagic	Allows inclusion of special characters in search patterns without a preceding backslash
nowrapscan	Prevents searches from wrapping around the end or beginning of a file.
mesg	Permits display of messages sent to your terminal with the **write** command

Summary of print commands

Table 2-11 Print commands

Command	Description
lp(C)	Sends a file or files to the printer
cancel(C)	Cancels a print request
lpstat(C)	Checks the status of a print job

Summary of file processing utilities

Table 2-12 File processing utilities

Utility	Description
diff(C)	Compares the contents of two files
sort(C)	Alphabetizes lines in a file
uniq(C)	Removes adjacent lines that are repeated in the file
grep(C)	Selects and extracts lines from a file
wc(C)	Counts words in a file

Chapter 3

Communicating using mail

The UNIX system **mail** application is a versatile communication facility that allows users to compose, send, receive, forward, and reply to mail. Users can also create distribution groups to send copies of a message to multiple users. These functions are integrated so that all users can quickly and easily communicate with each other.

This chapter is organized to satisfy the needs of both novice and advanced users. The first section discusses basic **mail** concepts. The second section provides demonstrations of the most commonly used commands. Later sections describe the more advanced **mail** commands and some advanced uses of **mail**.

For a quick introduction to get started using **mail** immediately, see the *Tutorial*. For a complete list of **mail** functions, refer to **mail**(C) in the *User's Reference*.

You should be familiar with the use of an editor such as **vi** in order to conveniently create mail.

Basic concepts

It is much easier to use **mail** if you understand the basic concepts that underlie it. The concepts discussed in this section are:

- mailboxes
- messages
- modes
- headers
- command syntax
- message lists

Mailboxes

It is useful to think of the **mail** system as modeled after a typical postal system. What is normally called a post office is called the "system mailbox" in this chapter. The system mailbox contains a file for each user in the directory */usr/spool/mail*. Your own personal or "user mailbox" is the file named *mbox* in your home directory. Mail sent to you is put in your system mailbox and is automatically saved in your user mailbox after you have read it. Note that the user mailbox differs from a real mailbox in these respects:

- The user mailbox is *not* the place where mail is initially routed—that place is the system mailbox in the directory */usr/spool/mail.*
- Mail is not picked up *from* your user mailbox.

Messages

In **mail**, the message is the basic unit of exchange between users. Messages consist of two parts: a heading and a body. The heading contains the following fields:

To:	This field is mandatory. It contains one or more valid user names to which you can send mail.
Subject:	This optional field contains text describing the message.
Cc:	The carbon copy field contains one or more valid names of those who are to receive copies of a message. Message recipients see these names in the received message. This field is optional.
Bcc:	The blind carbon copy field contains the one or more valid names of people who are to receive copies of a message. Recipients do *not* see these names in the received messages. This field is optional.

The body of a message is the text you enter exclusive of the heading. The body can be empty.

Modes

The **mail** program has two main modes: *compose mode* and *command mode.* You create a message in compose mode. In command mode, you perform **mail** operations for managing your mail.

The most common way of using **mail** is to begin a session by entering:

mail

If you have mail waiting, this command automatically places you in command mode. In this mode, you can enter commands for handling your mail. If you have no mail waiting, you see the following message and you return to the UNIX shell:

```
No mail in /usr/spool/mail/login
```

From the shell or from **mail** command mode, you can enter compose mode to create a message with:

mail ***username***

where ***username*** is the user name of the person to whom you want to send mail. In compose mode, you can enter the text of your message ending each line with a ⟨Return⟩. Send the message by pressing ⟨Ctrl⟩d on a new line; you then exit from the **mail** program and return to the UNIX shell. From compose mode, you can issue commands called *compose escapes* that allow you to temporarily leave or escape from compose mode. Compose escapes, which must be entered at the beginning of a line, begin with a tilde (~) and so are also called *tilde escapes.*

Once you press ⟨Return⟩ to end a line of a message you are creating, you cannot change that line from within compose mode. You must enter *edit mode* to change the line. In edit mode, you edit the body of a message using the full capabilities of an editor.

To enter edit mode from compose mode, use the compose escape **~e** to enter **ed**, the line editor, or **~v** to enter **vi**, the visual editor. It is often useful to be able to invoke either a line or visual editor, depending on the type of terminal you are using. When you finish editing the message, write it out and quit the editor; **mail** responds with:

```
(continue)
```

You are now back in compose mode and can continue creating your message.

You can also enter edit mode from command mode to edit any existing message. Use either **ed** or (**vi**) to do this. When you save the message and quit the editor, **mail** reads the message back into the message buffer.

If you want to mail a message that already exists in a file, you can do so from the UNIX system command line (without entering **mail**) as follows:

```
mail john < letter
```

Here, the file *letter* is sent to the user *john*.

NOTE Be very careful when mailing a file with the input redirection symbol (<). If you accidentally enter the output redirection symbol (>), you overwrite the file, destroying its contents.

When invoking **mail** from the shell, certain **mail** command-line options are available. Two useful command-line options are the **-s** (subject) option and the **-c** (carbon copy) option. You can specify a subject and carbon copy recipients on the command line with these options. For example, you could send a file named *note* with the *Subject:* "Important Meeting" by entering the following command:

```
mail -s "Important Meeting" -c "ted bob" bill joe sue < note
```

The *To:* field contains *bill, joe,* and *sue*; the *Cc:* field contains *ted* and *bob.*

All command-line options must appear before the list of users for the *To:* field. If an argument to an option contains multiple words, the entire argument set must be enclosed in quotes. Other command-line options are described in the **mail**(C) manual page.

Headers

When you enter **mail** command mode, a list of message headers is displayed that looks something like this:

```
  N  3 john Wed Sep 21 09:21 26/91 "Notice"
> N  2 sam  Tue Sep 20 22:55  6/91  "Meeting"
  U  1 tom  Mon Sep 19 01:23  6/91  "Invite"
```

By default, **mail** displays headers in reverse chronological order, the most recent message is displayed at the top of the list. The messages are numbered in ascending order from first received to most recently received; the message at the top of the list has the highest number. You can change the order in which headers are displayed by setting the **chron** and **mchron** options.

A header is a single line of text containing descriptive information about a message. (Note that we use the word *heading* to describe the first part of a message, and *header* to describe **mail**'s one-line description of a message.) The header contains:

- a greater-than sign (>) pointing to the current message
- a status indicator: "N" for new and "U" for unread
- the number of the message
- the sender
- the date sent
- the number of lines and characters
- the subject (if the message contains a *Subject:* field)

Message headers are displayed a screenful at a time. You can set the size of a screen with the **screen** option. You can move forward one screenful with the **headers (h)** command:

```
h +
```

You can move backward one screenful with **h -**. Both plus and minus take an optional numeric argument that indicates the number of header windows to move forward or backward before printing. With no argument at all, the **headers** command displays a window of headers in which the header of the current message is at the center.

The following are some characteristics of the header list:

- Deleted messages do not appear in the listing.
- Messages saved with the **save** or **write** command are flagged with a star (*).
- Messages selected with the **mbox** command to be saved in your user mailbox are flagged with an "M."
- Messages held in the system mailbox with the **hold** or **preserve** command are flagged with a "P."

Command syntax

Each **mail** command has its own syntax. Some commands take no arguments, some take only one, and others take several arguments.

Each **mail** command is entered on a line by itself, and any arguments follow the command word. The command need not be entered in its entirety; you can use its unique abbreviation. For example, you can enter "s" instead of "save" for the **save** command, "se" instead of "set" for the **set** command. Throughout this chapter, the appropriate abbreviation is enclosed in parentheses after the name of the command.

After you enter the command itself, you should enter one or more spaces to separate the command from its arguments. If a **mail** command does not take arguments, any arguments you give are ignored and no error occurs.

Message lists

Many **mail** commands take a list of messages as an argument. A *message list* is a list of message identifiers, ranges, users, search strings, or message types separated by spaces or tabs. For commands that take a message list as an argument, if no message list is given, the current message is used.

Message identifiers can be either decimal numbers, which directly specify messages, or one of three special characters: ^ (caret), . (dot), and $ (dollar sign), which specify the first, current, and last *non-deleted* messages, respectively.

A range of messages is two message identifiers separated by a dash. To display the headers of all the messages from the current message to the last message, enter:

```
h .-$
```

By giving a user name as part of a message list, you can display the messages sent by a particular user. For example, if you want to read only the messages sent by your manager, enter:

```
p markt
```

The **print** (**p**) command displays those messages on the screen one after another.

You can use a search string to specify all messages with the given string in the *Subject:* line (case is ignored). For example, to display the headers of only the messages with "meeting" in the *Subject:* line, enter:

```
h /meeting
```

You can create a message list by defining the type of messages in which you are interested. Use a colon followed by one of the following key letters:

d	deleted messages
n	new messages
o	old messages
r	read messages
u	unread messages

For example, to see a list of headers of the messages you deleted, enter:

```
h :d
```

Message lists can contain combinations of numbers, ranges, and names. For example, to delete all messages about your print jobs from user *lp* that are numbered from the first non-deleted message to 7 or 11 and 12, use the **delete** (**d**) command with the following message list:

d lp ^-7 11 12

As a shorthand notation, you can specify an asterisk (*) to mean all non-deleted messages. For example, to completely clean out your mailbox, use the **save** (**s**) command with an asterisk and a filename to save all non-deleted messages to the specified file:

s * mail.old

The asterisk symbol cannot be used with any other message list notation.

Using mail

This section demonstrates some of **mail**'s more commonly used features. Refer to the **mail**(C) manual page for details about other commands.

Composing and sending a message

Try sending a message to yourself by entering the following command from UNIX command level:

mail ***username***

where ***username*** is your login name.

If the **asksub** option is set, **mail** prompts for a *Subject:* line.

```
Subject: ▌
```

Enter a one-line summary of the message, then press ⟨Return⟩ to enter compose mode.

In compose mode, the text that you enter is appended one line at a time to the body of the message you are sending. Normal line editing functions are available when entering text, including ⟨Ctrl⟩**u** to kill a line and ⟨Bksp⟩ to back up one character.

Next, enter the following lines. Press ⟨Return⟩ at the end of each line.

This is a message sent to myself.
I compose a message by entering lines of text.

Press <Ctrl>d on a new line to end the message.

To view the message you are composing (including the heading fields) as it will appear when you send it, enter:

~p

This displays the following:

```
-------
Message contains:
To: login
Subject:  Sample Message

This is a message sent to myself.
I compose a message by entering lines of text.
Press (Ctrl)d on a new line to end the message.
(continue)
█
```

You can abort a message you are composing by entering two interrupts in a row (i.e., pressing INTERRUPT twice). In this case, the message is not sent. When you abort a message, a copy of the body of the undelivered message is appended to the file *dead.letter* in your home directory.

When you are ready to send your message, press (Ctrl)d on a line by itself to end the message and to send it. Once you have sent mail, there is no way to undo the act, so be careful.

If mail cannot be delivered to the address you specified, you are notified with a message that includes the undeliverable message.

Reading your mail

The message you sent yourself should have arrived in your system mailbox. To begin a **mail** session, enter:

mail

mail then displays a sign-on message and a list of message headers:

```
SCO Mail version 4.1  Type ? for help.
"/usr/spool/mail/login": 1 message
> N  1 login    Fri Aug 31 12:26  9/229 "Sample Message"
?█
```

The question-mark prompt prompts you to enter a **mail** command. You can set the prompt to a different string with the **prompt** option. To get help on all the available **mail** commands, enter:

?

To display the message that you sent to yourself, press ⟨Return⟩. **mail** displays:

```
From login Fri Aug 20 12:26:52 1985
To:  login
Subject:  Sample Message

This is a message sent to myself.
I compose a message by entering lines of text.
```

The message you sent to yourself now contains information about the sender of the message — a line telling who sent the message and when it was sent. The next line tells who the message was sent to. If a *Subject:* or a carbon copy (*Cc:*) field was specified by the sender, they too are displayed when you read the message.

You can configure your environment so that you are notified whenever new mail is sent to you even if you are not in **mail**. To do so, you should set the **MAIL** shell variable if you are using the Bourne shell or the **mail** shell variable if you are using the C-shell. For more information, see "The Bourne shell" and "The C shell" chapters of this guide and **csh**(C) in the *User's Reference.*

After examining a message, you will most likely want to either leave the message in your system mailbox, save it in a file, reply to it, or delete it. These and other useful **mail** operations are described in the next sections.

Saving a message

Sometimes you want to save a message for future reference. If you leave **mail** with the **quit** (**q**) command without performing any other operation on your message, the message is normally saved in the user mailbox. When you quit, **mail** displays the following message before returning you to the UNIX shell:

```
Saved 1 message in /u/login/mbox
```

To keep the message in the system mailbox, use the **hold** or **preserve** command; **mail** displays the following message:

```
Held 1 message in /usr/spool/mail/login
```

You see the same message the next time you invoke **mail.**

Saving many messages in the user or system mailbox can be confusing and can slow down processing. You can use the **save (s), write (w),** or **Write (W)** commands to organize your mail by putting messages that relate to each other in a specific file.

If you use **save, mail** saves the message to the name of the mail folder given as the last argument on the command line. For example, the following command appends the current message to the file *letters*:

```
s letters
```

Each saved message is marked with an asterisk (*). When you quit from **mail,** saved messages are normally deleted from the system mailbox.

If the file *letters* does not already exist, **save** creates it. The **save** command writes the entire message, including the header fields (such as *To:*, *Subject:*, and *Cc:*) and the ⟨Ctrl⟩**a** delimiters between messages to the specified filename. **mail** now treats the file *letters* as a "mail folder." (For more information about the different mail folder formats, see the section "Converting mailboxes to the new MMDF format" later in this chapter.)

You can access messages saved in a mail folder by specifying the filename with the **mail -f** command-line option or with the **folder (fold)** or **file (fi)** command from within **mail.** Both of these methods read in the specified file, giving you access to the messages in that folder in the same way you have access to the messages in your system mailbox when you invoke **mail** normally. Your user mailbox is also a mail folder; its messages can be accessed in the same way.

NOTE If you leave a mail folder by switching to another folder or back to your system or user mailbox, you can no longer use **undelete** to restore a deleted message from the original folder.

If you want to save your message text and headers without the ⟨Ctrl⟩**a** delimiters (in other words, you do not want to be able to access the messages using **mail**), use the **Write (W)** command. As with **save, Write** marks the message as "saved" (with an asterisk) and the message is deleted from your system mailbox when you leave **mail.**

The **mail** command also provides the **write (w)** command for saving messages without the headers (or ⟨Ctrl⟩**a** delimiters) to a plain text file. In this case, **write** keeps only the message text, no headers.

Printing a message

Another way to save a message is with the **lpr** (**l**) command, which sends the message to the lineprinter. This command takes a message list as its argument, then paginates and prints each message on the lineprinter. For example:

l doug

prints each message from the user *doug*.

Replying to a message

Often, you want to deal with a message by responding to its author right away. You can use the **reply** (**r**) command to set up a response to a message, automatically addressing a reply to the person who sent the original message. The original message's *Subject:* field is copied as the reply's *Subject:*. Each message is created in compose mode; thus, all compose escapes work, and you terminate messages by pressing 〈Ctrl〉**d**.

The **Reply** (**R**) command works just like its lowercase counterpart, except that copies of the reply are also sent to everyone shown in the original message's *To:* and *Cc:* fields.

Deleting a message

Unless you indicate otherwise, each message you receive is automatically saved in the user mailbox when you quit **mail**. Often, however, you do not want to save messages you have received. To delete a message, use the **delete** (**d**) command. For example:

d1

prevents **mail** from retaining message 1 in the user mailbox. The message disappears altogether, along with its number.

The **dp** command deletes the current message and displays the next message; this command is useful for quickly reading and disposing of mail.

The **undelete** (**u**) command causes a message that has been previously deleted with **d** or **dp** to reappear as if it had never been deleted. For example, to undelete message 1, enter:

u1

You cannot undelete messages from previous **mail** sessions; they are permanently deleted.

Forwarding mail

To forward a copy of a message, use the **forward** (**f**) command. This causes a copy of the current message to be sent to the specified users. For example, to forward the current message to someone whose login name is *john*, enter:

```
f john
```

John receives the forwarded message, along with a heading showing that you forwarded it. The forwarded message is indented one tab stop inside the new message. You can also give an optional message list:

```
f 2 6 john
```

This forwards messages 2 and 6 to *john*.

The **Forward** (**F**) command is identical to the lowercase **forward** command, except that the forwarded message is not indented.

Executing shell commands

You can execute a shell command without leaving **mail** from either **mail** command mode or compose mode. From command mode, precede the command with an exclamation point. For example:

```
!date
```

This command displays the current date without leaving **mail**.

From compose mode, precede the command with ~!. The command is executed, and you return to **mail** compose mode without altering your message.

From command mode, you can enter a new shell with the **shell** (**sh**) command. To exit from this new shell and return to **mail** command mode, press 〈Ctrl〉**d**.

Sending mail to remote sites

You can send mail to users on remote computer sites that are networked to your own site with UUCP. To find out which UUCP sites your computer communicates with, enter the following command at the UNIX prompt:

```
uuname
```

A list of site names is displayed.

To send mail to a user on a UUCP site, enter the following command:

mail ***site-name*!*login***

The site name must be followed by an exclamation point (!).

For example, to send mail to user *markt* on site *bowie*, enter the following command:

```
mail bowie!markt
```

Proceed to use **mail** just as if you were mailing to a local user.

You can enter several site names on a command line; be sure to follow each one with an exclamation point. As another example, suppose your site talks to UUCP site *bowie* and that *bowie* talks to UUCP site *bradley*. You can send mail to user cindy on *bradley* by entering the following command:

```
mail bowie!bradley!cindy
```

NOTE If you are using the C-shell, you must "escape" exclamation points with the backslash (\). A C-shell user enters the command above as follows:

```
mail bowie\!bradley\!cindy
```

For more information on communicating with remote sites, see the "Communicating with other sites" chapter in this guide.

Leaving mail

When you finish reading all your messages, you can leave **mail** by entering the **quit** (**q**) command. All messages are held in your user mailbox, except the following:

- deleted messages, which are discarded irretrievably
- messages marked with the **hold** or **preserve** command (these messages are saved in your system mailbox)
- if the **hold** option is set, messages that you have read are automatically saved in your system mailbox
- messages saved with the **save** or **write** commands

Forwarded messages are *not* removed from the system mailbox.

If you want to leave **mail** quickly without altering either your system or user mailbox, you can use the **exit** (**x**) command. This returns you to the shell without changing anything: no messages are deleted or saved in your user mailbox.

Leaving compose mode temporarily

While composing a message to be sent to others, you might need to change heading fields, invoke the text editor on a partial message, execute a shell command, or perform some other useful function. **mail** provides these capabilities through *compose escapes*. These escapes consist of a tilde (~) at the beginning of a line, followed by a one- or two-character command that specifies the function to be performed.

To get a list of the available compose escapes, enter the following command from compose mode:

~?

These compose escapes are available *only* when you are composing a new message; they have no meaning when you are in **mail** command mode. The **mail**(C) manual page contains details about these escapes.

Editing headers

To add additional names to the list of message recipients, enter the following escape:

~t ***login1 login2 ...***

You can name as many additional recipients as you like. Note that users originally on the recipient list still receive the message; you cannot remove anyone from the recipient list with **~t**. To remove a recipient, use the **~h** escape discussed later in this section.

You can replace or add a *Subject:* field by using the **~s** escape:

~s ***line-of-text***

This replaces any previous subject with ***line-of-text***. The subject, if given, appears near the top of the message, prefixed with the heading *Subject:*. You can see what the message looks like by using **~p**; this displays all heading fields along with the body of the text.

You might occasionally prefer to list certain people as recipients of carbon copies of a message rather than direct recipients. The following escape adds the named people to the *Cc:* list:

~c ***login1 login2 ...***

Similarly, the following escape adds the named people to the *Bcc:* (Blind carbon copy) list.

~b ***login1 login2 ...***

The people on this list receive a copy of the message, but are not mentioned anywhere in the message you send.

The recipients of the message are given in the *To:* field; the subject is given in the *Subject:* field, and carbon copy recipients are given in the *Cc:* field. If you want to edit these in ways impossible with the **~t**, **~s**, and **~c** escapes, you can use:

~h

where "h" stands for "heading." The escape **~h** displays *To:* followed by the current list of recipients and leaves the cursor at the end of the line. If you enter ordinary characters, they are appended to the end of the current list of recipients. You can also use the normal UNIX command-line editing characters to edit these fields, so you can erase existing heading text by backspacing over it.

When you press ⟨Return⟩, **mail** advances to the *Subject:* field, where the same rules apply. Another ⟨Return⟩ brings you to the *Cc:* field, and another brings you to the *Bcc:* field. Each of these fields can be edited in the same way. Finally, another ⟨Return⟩ leaves you appending text to the end of your message body. Remember that you can always use **~p** to see what the message looks like.

Adding a file to the message

It is often useful to be able to include the contents of some file in your message. Use the following escape to append the named file to your current message:

~r ***filename***

mail complains if the file does not exist or cannot be read. If the read is successful, **mail** displays the number of lines and characters appended to your message.

The following escape reads in the file *dead.letter* in your home directory:

~d

This is often useful because **mail** normally copies the text of your message buffer to *dead.letter* whenever you abort the creation of a message. You can abort the message by entering two consecutive interrupts or by entering a **~q** escape.

Enclosing another message

If you are sending mail from within **mail**'s command mode, you can insert a message, that was previously sent to you, into the message that you are currently composing. For example, enter:

~m 4

This reads message 4 into the message you are composing, shifted right one tab stop. Use the following escape to perform the same function, but with no right shift:

~M 4

You can name any non-deleted message or list of messages.

Setting up your environment

You can define your **mail** environment with switch and string options that can be set with the **mail** commands **set** and **unset**. A switch option is either on or off (set or unset). String options are strings of characters that are assigned values with the syntax *option=string*. Multiple options can be specified on a line. For example, in the following **set** command:

set dot askcc SHELL=/usr/bin/sh

the options **dot** and **askcc** are switch options; **SHELL** is a string option.

The **set** command with no arguments displays the options currently set.

You can create a personal mailing list with the **alias (a)** command. By using an alias, you can send mail to one name and have it go to a group of people. With no arguments, **alias** displays all currently defined aliases. With one argument, it displays the users defined by the given alias.

It is most useful to place **set, unset,** and **alias** commands in the file *.mailrc* in your home directory. The *.mailrc* file defines your personal default environment when you invoke **mail**. Whenever **mail** is invoked, it first reads the file */usr/lib/mail/mailrc,* then the file *.mailrc* in the your home directory. System-wide **set** options and system-wide aliases are defined in */usr/lib/mail/mailrc.* These are installed by the system administrator on your system. Personal aliases and personal **set** options are defined in *.mailrc.*

The following is a sample *.mailrc* file:

```
# number sign introduces comments
# personal aliases office and cohorts are defined below
alias office bill joe sue
alias cohorts john mary bob beth mike
# set dot lets messages be terminated by period on new line
# set asksub prompts for Subject: before entering compose mode
set dot asksub
# changes to always begin executing from the same directory
cd
```

The following sections demonstrate how to create mailing lists and describe a few of the common **set** options. Refer to the **mail(C)** manual page for details about other options.

Creating mailing lists

The **alias** command links a group of names with the single name given by the first argument, thus creating a mailing list. For example, you can create an alias called *beatles* with the following command:

alias beatles john paul george ringo

Now, whenever you used the name *beatles* in a destination address (you enter **mail beatles**), **mail** expands it to the four names aliased to *beatles*.

Aliases are expanded in mail sent to others so that they can **Reply** to each individual recipient. For example, the *To:* field in a message sent to *beatles* reads:

```
To:  john paul george ringo
```

and not:

```
To:  beatles
```

Keeping mail in the system mailbox

The **hold** option determines whether messages remain in the system mailbox when you exit **mail**. If you do not set **hold**, the examined messages are automatically placed in the *mbox* file in your home directory (your user mailbox). They are *removed* from the system mailbox when you quit.

The Cc: prompt

The **askcc** switch causes prompting for additional carbon copy recipients when you finish composing a message. Responding with a ⟨Return⟩ signals your satisfaction with the current list. Pressing INTERRUPT displays:

```
interrupt
(continue)
█
```

so that you can return to edit your message.

Listing messages in chronological order

The **chron** switch causes **mail** to list and display messages in chronological order. By default, messages are listed and displayed with the most recent first. Set **chron** when you want to read a series of messages in the order you received them.

The **mchron** switch, like **chron**, displays messages in chronological order, but lists them in the opposite order, that is, highest-numbered, or most recent, first. This is useful if you keep a large number of messages in your mailbox and you want to list the headers of the most recently received mail first but read the messages themselves in chronological order.

Using advanced mail features

This section discusses advanced features of **mail**; these features are useful for people with some existing familiarity with the **mail** system.

Using mail as a reminder service

Besides sending and receiving mail, you can use **mail** as a reminder service. Several UNIX system commands have this idea built in to them. For example, the **lp** command's **-m** option causes mail to be sent to the user after files have been printed on the lineprinter. When you log in, the operating system automatically examines the file named *calendar* in your home directory and looks for lines containing either today or tomorrow's date. These lines are sent to you by **mail** as a reminder of important events.

If you program in the shell command language, you can use **mail** to signal the completion of a job. For example, you might place the following two lines in a shell procedure:

```
biglongjob
echo "biglongjob done" | mail login
```

You can also create a logfile that you want to mail to yourself. For example, you might have a shell procedure that looks like this:

```
dosomething > logfile
mail login < logfile
```

For information about writing shell procedures, see "The Bourne shell" chapter in this guide.

Redirecting incoming messages to other folders

SCO UNIX System V/386 uses the MMDF Mail Transport Agent (MTA). You can use MMDF features to redirect specific messages to folders other than your system mailbox. Redirecting mail is useful for keeping all the mail on a particular subject or the messages sent to an alias to which you belong in one folder so that you can read them all at one time.

To tell MMDF to deliver mail to different folders, set up the special *.maildelivery* file in your home directory. Use the following procedure to set up *.maildelivery*:

1. Create the *.maildelivery* file in your home directory.
2. Make sure the permissions on the file are set to 644 (`-rw-r--r--`). To set the permissions, use the following command:

   ```
   chmod 644 $(HOME)/.maildelivery
   ```

3. Now, create a directory in which to put all the redirected mail. For example, create a directory called *spool* in your home directory:

   mkdir ~/spool

   NOTE This does not work in Bourne shell.

   If your redirected mail directory is something other than *spool,* change all occurrences of spool in the examples to the new directory name.

   NOTE If you do not create the directory in which to redirect your mail before setting up *.maildelivery,* you can create an infinite loop as MMDF tries to deliver mail to a non-existent directory. Make sure you redirect mail to an existing directory!

4. Add the following lines to *.maildelivery*:

```
# Header        Content  Action  Result  Filename
to              login    >       ?       /usr/spool/mail/login
cc              login    >       ?       /usr/spool/mail/login
resent-to       login    >       ?       /usr/spool/mail/login
apparently-to   login    >       ?       /usr/spool/mail/login
```

   The number sign (#) is a comment character. Replace ***login*** with your login name.

   These lines make sure all mail specifically addressed to you (mail with your name on the *To:*, *Cc:*, *Resent-To:*, or *Apparently-To:* lines) is delivered to your system mailbox file.

   The " > " character in the Action column tells MMDF to append the message to the specified file (*/usr/spool/mail/**login***). The " ? " in the Result column tells MMDF to consider the message "delivered" if the action is successful. In other words, if the message arrives safely in the system mailbox, MMDF considers it "delivered." Note that " ? " prevents MMDF from carrying out the specified action if the message has already been delivered. This is important if you don't want MMDF to deliver a message to more than one folder.

5. Use this step to redirect mail addressed to an alias to which you belong. For example, if you want all the messages sent to the *engr* alias to go to a folder called *~/spool/engr.mail,* add the following lines to *.maildelivery*:

```
to   engr   >   A   /u/login/spool/engr.mail
cc   engr   >   A   /u/login/spool/engr.mail
```

The "A" in the Result column is identical to the ? in step 3, except that, even if the message is already considered "delivered," MMDF still performs the specified action.

MMDF creates the *engr.mail* folder when it first delivers mail to that location; you do not need to create the folder for MMDF to deliver the messages.

6. This step explains how to redirect messages with specific patterns in the *Subject:* header. To do this, simply specify "Subject" in the Header column and the pattern in the "content" column. For example, to redirect mail with the word "Style" in the *Subject:* header to *~/spool/style.mail,* add the following line to *.maildelivery*:

```
subject    "Style"    >   A   /u/login/spool/style.mail
```

7. Finally, you must add a line to catch ana deliver all the messages that don't match the conditions you've specified. Add the following lines to *.maildelivery*:

```
default   -   >   A   /usr/spool/mail/login
#
```

The "default" in the Header column tells MMDF to match this field if the message hasn't already been delivered. The dash character (-) is simply a placeholder for an existing but unused field.

See the section, "Saving a message," for information on using the **folder** command to access the messages that you redirect to other folders.

Sending automatic responses when on vacation

MMDF includes a utility called **rcvtrip** that you can use to generate automatic replies to the messages that you receive while you are on vacation. MMDF still delivers your mail to your system mailbox (or other folders, if you've set up *~/.maildelivery* to redirect messages).

NOTE The **rcvtrip** utility sends an automatic response only to messages that specifically include your login name on the *To:* or *Cc:* lines.

In addition, **rcvtrip** keeps a record of who you have already sent a response to, and does not send another even if they send you more than one message while you are gone.

Use the following procedure to set up the appropriate files so that **rcvtrip** replies automatically to your incoming mail:

1. First, create a file called *tripnote* in your home directory. Make sure the permissions on the file are set to 644 (`-rw-r--r--`). To set these permissions, use the following command:

   chmod 644 ~/tripnote

2. Edit *~/tripnote* and add the response that you want MMDF to send to people when they send you mail. For example, you might want to tell people about where you are and when you will be back.
3. Create an empty file called *triplog* in your home directory using **touch**(C):

   touch ~/triplog

   MMDF uses this file to keep track of the people to whom you have already sent automatic responses.
4. Make sure that the permissions on *~/triplog* are set to 644:

   chmod 644 ~/triplog

5. Now, add the following as the first line in your *~/.maildelivery* file:[1]

   ```
   *  -  pipe  Rrcvtrip $(sender)
   ```

   NOTE You do not need to set up *.maildelivery* for redirecting messages if you just want to use **rcvtrip**. If this is the case, simply create the *.maildelivery* file in your home directory, add the **rcvtrip** line above, and make sure the permissions are set to 644.

   This line tells MMDF to pipe all your messages to the **rcvtrip** program but does not consider the messages "delivered." In other words, MMDF still delivers the incoming messages to the appropriate folders. (See the **rcvtrip**(C) manual pages for details.)

   You should leave this line commented out (use the number (#) character) until immediately before you leave for your vacation.
6. When you return, comment out the **rcvtrip** line in your *.maildelivery* file.

The *triplog* file contains a list of all the people that sent you mail when you were gone. These lines list the address, date, and time you received the message, part of the *Subject:* line, and a plus (+) or minus (-) character to indicate whether or not **rcvtrip** sent an automatic response. You can use this file to check on who received automatic messages.

1. For information on setting up your *.maildelivery* file to redirect messages to other folders, see the previous section, "Redirecting incoming messages to other folders."

Forwarding your messages to another person

MMDF includes a utility called **resend**(C) that allows you to forward all your incoming messages to another person automatically. For example, you might want someone else to read your mail when you go on vacation to ensure that any time-critical mail is answered promptly. As with **rcvtrip**, you use **resend** by adding a line to your *.maildelivery* file.[2]

Use the following steps to set up the *.maildelivery* file so that **resend** forwards your messages automatically:

1. If you have not already done so, create a file called *.maildelivery* in your home directory and make sure the permissions are set to 644.
2. Add the following as the first line in your *~/.maildelivery* file:

```
To   login   |   A   /usr/bin/resend   user_2
```

   Replace ***login*** with your address and ***user_2*** with the address of the person to whom you want to forward your messages.

This line in *.maildelivery* tells MMDF to redirect any mail with ***login*** in the *To:* header (mail addressed to you) to the **resend** utility which then forwards the messages to ***user_2***.

For more information on the format of the *.maildelivery* file, see the **maildelivery**(F) manual page or the section, "Redirecting incoming messages to other folders," earlier in this chapter. See **resend**(C) for more information on the **resend** utility.

Converting mailboxes to the new MMDF format

If you are converting to SCO UNIX System V/386 from XENIX System V or another UNIX system, your old folders are in the XENIX-style (older UNIX) format. XENIX format uses the "From<space>" lines to delimit between messages in mail folders; MMDF uses ⟨Ctrl⟩**a** characters.

By default, the **mail**(C) program maintains the current format of every mail folder that you read with this program. For example, if a mail folder is in MMDF format (with ⟨Ctrl⟩**a** delimiters), **mail** uses this format when adding messages to the folder. If you use **mail** to read a folder that has not been converted to MMDF format, it prompts you to convert the folder.

2. For information on setting up your *.maildelivery* file to redirect messages to other folders or to call **rcvtrip**, see the sections, "Redirecting incoming messages to other folders," and "Sending automatic responses when on vacation" earlier in this chapter.

If you use a Mail User Agent (MUA) other than **mail**, you should convert your old folders to use the new MMDF format using the **cnvtmbox**(ADM) utility. To do this, run the following commands for each folder:

```
/usr/mmdf/bin/cnvtmbox old_folder new_folder
mv new_folder old_folder
```

For more information, see the **cnvtmbox**(ADM) manual page.

Handling large amounts of mail

Eventually, you will face the problem of dealing with an accumulation of messages in your user mailbox. There are a number of strategies that you can employ to solve this problem concerning space in your mailbox file. Keep in mind the dictum: *When in doubt, throw it out.*

This means that you should only save *important* mail in your user mailbox. If your mailbox file becomes large, you must periodically examine its contents to decide whether messages are still relevant. To save space, consider summarizing very long messages.

The previously mentioned measures are not always helpful enough in organizing the many messages that you are likely to receive. Another effective approach is to save mail in files organized by sender, by topic, or by a combination of the two. However, be forewarned — this approach to organizing mail quickly eats up disk space. Using mail folders is described in "Saving a message."

You can create a directory to hold your mail folders and define that directory to **mail** with the **folder** option. Then, whenever you save a message without giving a pathname, **mail** puts the message in a file (or folder) in that directory. For example, if you want to save your messages by default in the directory *mail* in your home directory, use:

```
set folder=mail
```

If you forget the names of your mail folders, you can use the **folders** command to display the names of the files in the directory set by the **folder** option.

Summary

The tables in this section summarize the **mail** commands and variables discussed in this chapter.

Table 3-1 mail commands

Command	Abbreviation	Description
!		Executes a shell escape from command mode
alias	a	Creates personal mailing lists
delete	d	Deletes messages
dp		Deletes the current message and displays the next one
edit	e	Edits an existing message from command mode
exit	x	Exits **mail** without updating the folder
file	fi	Switches folders
folder	fold	Switches folders
forward	f	Forwards a message (indented) to another person
Forward	F	Forwards a message (not indented)
headers	h	Moves forward one screenful
lpr	l	Sends messages to the lineprinter
preserve		Holds messages in system mailbox
print	p	Displays non-deleted messages
quit	q	Updates the folder and quits **mail**
reply	r	Sends a reply to the author only
Reply	R	Sends a reply to everyone on the distribution list
save	s	Saves messages to folders
set		Sets **mail** options
shell	sh	Enters a new shell
undelete	u	Restores deleted messages
unset		Unsets **mail** options
visual	v	Edits an existing message from command mode
write	w	Writes messages to files

Table 3-2 Compose mode commands

Command	Description
~!	Executes a shell escape from compose mode
~?	Lists compose mode escapes
~b	Adds or changes the *Bcc:* header
~c	Adds or changes the *Cc:* header
~d	Reads in the *dead.letter* file
~h	Adds or changes all the headers
~m	Reads a message (indented) into the current message composition
~M	Reads a message (not indented) into the current message composition
~q	Aborts the current message composition
~r	Appends a file to your current message
~s	Adds or changes the *Subject:* header
~t	Adds names to the *To:* list

Table 3-3 mail options

Option	Description
askcc	Prompts for carbon copy recipients
asksub	Prompts for a *Subject:* line
chron	Displays messages in chronological order, most recent last
folder	Defines the directory to hold your mail folders
hold	Keeps messages in the system mailbox when you quit
mchron	Displays messages in chronological order, most recent first
prompt	Sets the prompt to a different string
screen	Sets the size of a screen

Table 3-4 Other commands

Command	Description
chmod	Changes the permissions on a file or directory
cnvtmbox	Converts old-style format mailboxes to MMDF format
mail	Sends a message or enter the **mail** utility
rcvtrip	Sends automatic responses to messages
touch	Creates and empty file
uuname	Determines which UUCP sites your computer communicates with

Table 3-5 Special files

File	Description
calendar	Contains reminders for the operating system to mail to you
.maildelivery	Redirects incoming mail to other folders
triplog	Keeps track of the people you send automatic responses to
tripnote	Contains the message for **rcvtrip** to send automatically

Chapter 4

Using disks and tapes

In general, the system administrator is responsible for creating backups of the filesystems on UNIX systems. However, you might find it useful to create backups of your own files as well periodically. The backup media used most often are floppy disks; however, you can also create backups on cartridge tapes. This chapter explains how to use the **tar**(C) command to create backups on disks and tapes. You can also use **tar** to copy files to disks or tapes so that you can transport them from one computer to another.

This chapter describes how to perform the following tasks:

- format floppy disks and mini cartridge tapes
- copy files to the backup media
- list the contents of the backup media
- extract files from the backup media
- make copies of floppy disks

Formatting disks and tapes

Before you create a backup, you must format the floppy disk(s) using the **format**(C) command. In general, cartridge tapes do not need to be formatted. However, tapes that are to be used with *mini tape drives* must be formatted.

This section explains how to format floppy disks and mini cartridge tapes.

Formatting floppy disks

Once you have formatted a floppy disk on a UNIX system, you can re-use it without reformatting it. If you have disks that have been formatted under a different operating system, you must reformat them on a UNIX system before they can be used with a UNIX system. However, be aware that floppy disks formatted under some operating systems cannot be used under other operating systems, even if they are reformatted.

The default floppy disk drive is determined at installation time by the media used to install the operating system. For example, if you installed the operating system using 5.25-inch disks, the default drive is the first 5.25-inch drive. If you installed the operating system using 3.5-inch disks, the default drive is the first 3.5-inch drive. This is important when using the **format** command without any arguments. For example, to format a disk in the default drive, enter the following command:

```
format
```

The system prompts you to insert a disk into the drive and to press ⟨Return⟩. The disk is then formatted using the default settings for formatting disks, as defined in */etc/default/format*. The default settings for 5.25-inch disks are **rfd096ds15**, where:

- r indicates a raw (character) interface to the disk (i.e. not buffered)
- fd0 is the drive number (0, 1, 2 or 3)
- 96 is the number of disk tracks per inch (48 or 96)
- ds is the number of sides (ds - double sides or ss - single side)
- 15 is the number of sectors per track (8, 9, or 15)

The default settings for 3.5-inch disks are **rfd0135ds18**. Most of these can be understood from the default settings for 5.25-inch disks, the only differences being the number of disk tracks per inch, which must be 135, and the number of sectors per track, which can be 9 or 18.

Here are a few examples that show how the configurations can be varied to format different types of disks. (The default drive is assumed in all cases.)

To format a 5.25-inch 360 kilobyte disk, enter:

```
format /dev/rfd048ds9
```

To format a 5.25-inch 720 kilobyte disk, enter:

```
format /dev/rfd096ds9
```

To format a 5.25-inch 1.2 megabyte disk, enter:

```
format /dev/rfd096ds15
```

To format a 3.5-inch 720 kilobyte disk, enter:

```
format /dev/rfd0135ds9
```

To format a 3.5-inch 1.44 megabyte disk, enter:

```
format /dev/rfd0135ds18
```

Formatting mini cartridge tapes

To format mini cartridge tapes, use the following command:

```
mcart format
```

Using the tar command

Use the **tar** command to save and restore files to and from an archive medium, typically a floppy disk.

The syntax of **tar** is:

tar [*options*] [*files*]

The *options* tell **tar** what you want to do; *files* are the files that you want to back up.

The *options* used most often with **tar** are as follows:

c	Creates a new backup, overwriting any files already on the backup destination.
x	Extracts files from backup media.
t	Lists the contents of backup media.
v	Displays the name of each file being processed.
f	Creates backups on a specified device.
u	Adds files to the backup if they are not already there, or if they have been modified since they were last written on the backup.

Creating backups with tar

The steps below explain how to back up all of the files in your home directory onto floppy disks. To back up a different directory, use **cd** to change to the new directory before using the **tar** command at step 3. To back up onto a tape, substitute the special device file associated with the tape, such as */dev/rctmini* or */dev/rct0*, for the floppy device file listed in these commands.

1. Log in on the console. This allows you to work within arm's reach of the floppy drive.

2. Determine how many floppy disks you need and format that many, using the **format** command as described in the earlier section "Formatting disk and tapes."

   To determine how many disks to format, use the **du**(C) command:

   du -s

   Your screen looks something like the following:

   ```
   du -s
   356
   ```

   This number represents the total number of 512-byte blocks used by the files in the current directory. In this example, to back up the directory, you need a total of 512 x 356 bytes, or roughly 183 kilobytes. Thus, you only have to format a single (1.2 megabyte) floppy disk to back up this directory.

   You can display the number of blocks for each individual file using the following command:

   du -s ***filename***

3. To back up all the files in you home directory, enter:

   ```
   tar cvf /dev/fd096ds15 .
   ```

   where **c** causes a new backup to be created (overwriting any files currently on the floppy disk), **v** causes each file to be displayed as the backing up takes place, and **f** causes the subsequent argument (**/dev/fd096ds15**) to be the destination of the backup.

   If **tar** requires more than 1 disk, the system prompts you to insert another "volume." Insert another disk and press ⟨Return⟩. When the shell prompt reappears, **tar** is finished backing up your files.

   To back up a single file onto a 1.2 megabyte disk, enter:

   ```
   tar cvf /dev/fd096ds15 ./filename
   ```

   Note that in this example, a dot and a slash (./) precede the ***filename***. This tells **tar** to treat ***filename*** as a "relative" rather than an "absolute" pathname (refer to the *Tutorial* for a definition of "relative" and "absolute" pathnames.

The **tar** command recreates all the subdirectories of the directory you are backing up on the backup media. Thus, if you have a */bin* directory in your home directory, **tar** creates a backup of it, and all the files in it, on the backup media.

Listing the contents of backups

To list the contents of tar-floppies (backups on floppies created with **tar**) use the **t** option. The examples in this section show how to list the contents of different types of disks in the primary floppy drive:

- To list the contents of a 5.25-inch 360 kilobyte tar-floppy:

  ```
  tar tvf /dev/fd048ds9
  ```

- To list the contents of a 5.25-inch 1.2 megabyte tar-floppy:

  ```
  tar tvf /dev/fd096ds15
  ```

- To list the contents of a 3.5-inch **tar** micro-floppy:

  ```
  tar tvf /dev/fd0135ds9
  ```

Extracting files from backups

To extract files from tar-floppies use the **x** option. The examples in this section explain how to extract files from different kinds of disks in the primary disk drive.

NOTE We recommend that you extract files from backup media into a temporary directory on the hard disk. Once in the temporary directory, you can use the **mv**(C) command to move the extracted files to the correct location on the filesystem.

The reason for using a temporary directory is that **tar** overwrites any files on the hard disk that have the same name as the file being extracted from the backup media. This can result in files being overwritten accidentally.

- To extract all of the files from a 5.25-inch 360 kilobyte tar-floppy:

  ```
  tar xvf /dev/fd048ds9
  ```

- To extract files from a 5.25-inch 1.2 megabyte tar-floppy:

  ```
  tar xvf /dev/fd096ds15
  ```

- To extract a single file from a 1.2 megabyte tar-floppy:

  ```
  tar xvf /dev/fd096ds15 ./filename
  ```

  Note that a dot followed by a slash (./) precedes *filename* in the example above. This assumes that you copied *filename* to the tar-floppy using the dot and slash format (./), as in the examples in "Creating backups with tar". This treats the *filename* as a "relative" pathname.

  When you copy files to a tar-floppy using this format, **tar** adds the ./ prefix to the filenames. Because you must enter a filename exactly as it appears on the floppy when extracting a file with **tar**, you must enter **./filename** if you copied *filename* to the tar-floppy using this format.

Try this **tar** command by inserting the disk containing the backup of your home directory (that you created in "Creating backups with tar") into the primary floppy drive and following these steps:

1. Enter the following command to change to the */tmp* directory:

   ```
   cd /tmp
   ```

2. Now, create a subdirectory in */tmp* by entering:

   ```
   mkdir username
   ```

   Replace ***username*** with your username.

3. Now, enter the following command:

   ```
   cd username
   ```

4. Use one of the following commands to extract a file from the tar-floppy:

   - If you are a Bourne shell user, and if your primary floppy drive is a 1.2 megabyte drive, try extracting a single file by entering the following command to extract *.profile*:

     ```
     tar xvf /dev/fd096ds15 ./.profile
     ```

   - If you are a C shell user, use the following command to extract *.login*:

     ```
     tar xvf /dev/fd096ds15 ./.login
     ```

   If your floppy drive is not a 1.2 megabyte drive, enter the appropriate special device filename.

5. To extract all of the files on a **tar** floppy, use one of the commands in step 4, but do not specify a file to extract.

6. To check that **tar** actually copied the files to the hard disk, enter the following command:

   ```
   lc -a
   ```

   (The **-a** option tells **lc** to show hidden files, those filenames that begin with a dot (**.**).)

Shorthand tar notation

The **tar** command also provides shorthand notation for referring to special device filenames. This notation allows you to specify numbers in place of full special device filenames. The file */etc/default/tar* assigns numbers to the various floppy and tape devices. Enter the following to display the contents of */etc/default/tar*:

> **more /etc/default/tar**

Your output should look similar to the following:

```
#  device                   block  size  tape
archive0=/dev/rfd048ds9     18     360   n
archive1=/dev/rfd148ds9     18     360   n
archive2=/dev/rfd096ds15    10     1200  n
archive3=/dev/rfd196ds15    10     1200  n
archive4=/dev/rfd096ds9     18     720   n
archive5=/dev/rfd196ds9     18     720   n
archive6=/dev/rfd0135ds18   18     1440  n
archive7=/dev/rfd1135ds18   18     1440  n
archive8=/dev/rct0          20     0     y
archive9=/dev/rctmini       20     0     y
#  The default device...
archive=/dev/rfd096ds15     10     1200  n
```

This file assigns **0** to the first 360 kilobyte drive, **1** to the second 360 kilobyte drive, **2** to the first 1.2 megabyte drive, **3** to the second 1.2 megabyte drive, and so forth.

The following examples show how to use the shorthand **tar** notation when creating and extracting backups on the primary drive:

- To copy all the files in the current directory to 5.25-inch 1.2 megabyte disks:

  tar cv .

  (The default device is */dev/rfd096ds15*. You do not have to specify the default device name in the command in order to use the default device.)

- To copy all the files in the current directory to 5.25-inch 360 kilobyte disks:

  tar cv0 .

- To extract *filename* from a 3.5-inch 720 kilobyte **tar** micro-floppy:

  tar xv4 *./filename*

Note that the version of */etc/default/tar* on your system might differ from our example; the system administrator might have modified it. For this reason, double-check the assignments in */etc/default/tar* on your system before you use this shorthand notation.

Copying disks

To protect against losing data stored on floppy disks, you can make copies of your floppy disks using the **diskcp**(C) command.

You must copy information onto formatted disks. The **diskcp** command allows you to format floppies before making copies.

Use the following steps to copy floppies on a system with two floppy disk drives:

1. Place the disk that you want to copy (the "source" floppy) in your primary floppy drive. If you created a backup of your home directory, as instructed in "Creating backups with tar", you can use it to follow this exercise.
2. Place another floppy (the "target" floppy) in the secondary drive. Note that any information already on the target disk will be destroyed.
3. To copy a 1.2 megabyte source floppy directly to a formatted target floppy, enter the following command:

   ```
   diskcp -d
   ```

   where the **-d** option indicates that the computer has dual floppy disk drives. To format the target floppy before copying, use this command instead:

   ```
   diskcp -d -f
   ```

4. Follow the instructions as they appear on your screen.

If you only have one floppy drive, the **diskcp** program copies the information from the source floppy to the computer's hard disk, prompts you to replace the source floppy with the target floppy, and then copies the information from the hard disk to the target floppy.

Use these steps to copy a floppy using only one floppy drive:

1. Place the source floppy in your floppy drive.
2. If the source floppy is a 1.2 megabyte floppy and you do not need to format the target floppy, enter the following command:

   ```
   diskcp
   ```

   To format the target floppy as a 1.2 megabyte floppy before copying, enter this command:

   ```
   diskcp -f
   ```

3. The **diskcp** command prompts you to remove the source disk from the drive and insert the target disk. Follow the instructions as they appear on your screen.

Now, place the target floppy in the primary floppy drive and verify that the files were copied correctly using **tar** to list the contents of the floppy:

```
tar tvf /dev/fd096ds15
```

If your floppy is a 360 kilobyte floppy, enter:

```
tar tvf /dev/fd048ds9
```

Note that you can use the shorthand **tar** notation in these commands, as explained in the previous section, "Shorthand tar notation."

Summary

UNIX systems provide utilities for creating backup copies of files, which you can store on floppy disks or cartridge tapes. Once you create a backup copy, you can display its contents, extract files from it, and create additional copies.

This chapter covers the following commands:

Table 4-1 Command review

Command	Description
format(C)	formats floppies and tapes
tar(C)	saves and restores files to and from floppies
du(C)	summarizes the current disk usage
diskcp(C)	creates copies of floppy disks

Chapter 5 Managing processes

This chapter explains the different ways you can manage your processes. It describes how to perform the following tasks:

- run processes in the background
- view the processes that you are running using the **ps**(C) command
- terminate jobs using the **kill**(C) command
- prioritize jobs with the **nice**(C) command
- specify the time you want to start executing your jobs using the job scheduling commands **at**(C), **batch**(C), and **crontab**(C)

> **NOTE** You must be authorized by the system administrator to use the **at**, **batch**, and **crontab** job scheduling commands.

Running jobs in the background

Normally, commands sent from the keyboard are executed consecutively. One command finishes executing before the next command begins. However, if you place a command in the background, you can continue entering commands in the foreground, even if the background command has not finished executing.

To place a command in the background, put an ampersand (&) at the end of the command line. For example, enter the following command to create, and then count, the characters in a large file.

```
cat /etc/termcap /etc/motd > largefile; \
wc -c largefile > characters &
```

NOTE Note that this command line is two lines long; to use a multiple-line command line, enter the backslash (\ ) before pressing ⟨Return⟩. The backslash tells the shell that the command line continues on the next line.

This command line runs the **cat** and **wc** commands in the background, displays a *job number,* and then returns you to the UNIX system prompt. You see something like this:

```
[1]  14145
```

When the job finishes, you see a message like the following:

```
[1] + Done cat /etc/termcap /etc/motd > largefile; \wc -c largefile; > ch
```

When you place commands in the background, you cannot abort them by pressing the INTERRUPT key, as you can with foreground commands. You must use the **kill**(C) command to abort a background process. The section, "Killing processes," describes how to use **kill**.

If you followed the exercise above, you should use **rm** to remove both *largefile* and *characters* when you have finished:

rm largefile characters

For more information about manipulating background jobs, see the section on "Using job control" in Chapter 10, "The Korn shell."

Checking the status of processes

At any time, you can check the status of your task using **ps**(C). The **ps** command lists all the "active" processes that you are running on the UNIX system (both in the background and foreground). The **ps** command is used as follows:

ps [*options*]

The following examples show you how to use the **ps** command; try them yourself:

```
$ ps (Return)
   PID TTY  TIME COMMAND
 21311 005  0:01 ksh
 29424 005  0:00 ps
$
```

In this example, the **ps** command prints information about the processes associated with the current terminal, in this case "005".

where:

PID	is the process identification
TTY	is the controlling terminal for the process
TIME	is the execution time for the process
COMMAND	is the command name

You can obtain the same information using the **-t *tlist*** option (where ***tlist*** is a terminal number), and specifying the terminal ID as in the following example:

```
$ ps -t 005
```

You can also use **ps** to print information about the processes belonging to a particular user. To do this, use the **-u *ulist*** option (where ***ulist*** is a username):

```
$ ps -u johnd
   PID TTY  TIME COMMAND
 21311 005  0:01 ksh
 29425 005  0:00 ps
$
```

In this example, **ps** prints information about the processes belonging to user *johnd*. Notice that the output of this command and the previous command is the same; this is because both commands are associated with the terminal identified as "005".

To obtain a listing with more process status information, use **ps** with the **-f** option:

```
$ ps -f (Return)
    UID   PID  PPID  C    STIME TTY  TIME COMMAND
  johnd 21311     1  0  Jul  5  005  0:01 -ksh
  johnd 29426 21311 13 10:24:16 005  0:00 ps -f
$
```

where:

UID	is the user ID of the process owner
PPID	is the process ID of the parent process
C	is the processor utilization for scheduling
STIME	is the starting time of the process

For more information about the options to **ps**, see the **ps**(C) manual page.

Killing processes

If you want to terminate one of your actively running processes, press ⟨Del⟩, ⟨Bksp⟩, or your equivalent INTERRUPT key on your keyboard, depending on your system setup. If you need to terminate a background process, or a process running on a different terminal, use the **kill**(C) command. The **kill** command terminates a process by sending it a *signal*. For more information about signals, see Chapter 8, "The Bourne shell."

The **kill** command without options terminates a process with signal 15, the software termination signal. The unconditional **kill** signal (-9) is more commonly used to terminate processes, because the receiving process cannot ignore it.

NOTE Unless you are the system administrator, you can only terminate your own processes.

In using **kill**, you specify which process to terminate by specifying its process ID (PID) on the command line.

When you first enter a background command, the system prints its process ID (PID) (also known as the "job number") on your screen. You can also get the process ID from the PID column output of the **ps** command.

The syntax for the **kill** command is as follows:

kill [*-signal_number*] *process_id* ...

The following example shows you how to use the **kill** command.

```
$ sleep 100 &
1408
$ kill 1408
1408: sleep: Terminated
$
```

In this example, you run the **sleep** process in the background; the system displays the job number, 1408. Then, you terminate the job using **kill**. The shell indicates when the job has been terminated.

The **sleep** command is normally used to delay the execution of another command, the delay being the number of seconds entered after the **sleep** command. For example:

```
sleep 10; who &
```

delays the execution of the **who** command by 10 seconds.

Prioritizing processes

If you do not need to execute your commands immediately, or if you have a large amount of processing to do, you can execute your command with a lower scheduling priority using the **nice**(C) command.

The syntax for **nice** looks like this:

nice [*-increment*] *command* [*arguments*]

The **nice** command allows you to change the priority of a job by specifying the "nice value" of between 20 and 39. The default nice value is 20.

When you use ***-increment*** to increase the nice value, you give the command a lower scheduling priority. For example, use **nice -5** to increase the nice value to 25.

NOTE The super user can run a job at a higher priority by setting the nice value between 0 and 20 using a double negative ***increment***. For example, **-10** sets the nice value to 10, giving the job a higher priority than the default nice value.

The following examples show you how to change the scheduling priority with **nice**:

```
$ nice -17 big_program
$
```

In this example, you execute ***big_program*** with a nice value of 37.

To run a program in the background and change the scheduling priority, use a command like the following:

```
$ nice -17 big_program &
1579
$
```

Here, you run ***big_program*** in the background with a nice value of 37.

Scheduling your jobs

This section explains how to use the **cron**(C), **at**(C), and **batch**(C) programs to schedule or delay the execution of programs (jobs).

The job-scheduling programs are as follows:

cron	Executes programs repeatedly at a specified time.
at	Delays the execution of programs until the time you specify.
batch	Delays the execution of programs until the system load is low (as determined by the system).

This section explains how you or the system administrator can use each of the job scheduling programs to automate regular operations or delay program executions that would otherwise slow down the system during peak usage.

Executing programs automatically

UNIX systems allow you to run programs automatically at specified times. Use the **cron** program to do this. The **cron** program allows you to run programs and perform tasks such as the following while you are out of the office:

- reminder messages
- file system administration
- time-consuming, user-written shell procedures
- cleanup procedures

Any task that you need to perform repeatedly at a specified time is a candidate for **cron** to run.

To use **cron**, you must first create a *crontab* file and then use the **crontab** command to submit this file to the */usr/spool/cron/crontabs* directory where **cron** looks to execute commands. The following sections explain how to do this.

Creating a crontab file

To use **cron** to run commands, you must first create a file to define the procedures that you want to run. This file is called the *crontab* file; however, you can name it anything that you want.

Each line in the *crontab* file defines one procedure. The line entry format looks like the following:

minute hour day month day-of-week command

Table 5.1 defines each of these fields.

Table 5-1 crontab fields

Field	Value
minute	numbers between 0 and 59
hour	numbers between 0 and 23
day	numbers between 1 and 31
month	numbers between 1 and 12
day-of-week	numbers between 0 and 6 (0 is Sunday)
command	command that you want to execute at the specified time

The following rules apply to the first five fields:

- Two numbers separated by a hyphen indicate a range of numbers between the two specified numbers.
- A list of numbers separated by commas indicates that you want to use only the numbers listed.
- An asterisk specifies all legal values.

For example, the following line indicates that you want the **reminder** command to run on the first and fourteenth of each month, as well as on every Tuesday:

```
0 0 1,14 * 2 reminder
```

If you place a percent sign (%) in the command field (sixth field), the operating system translates it as a newline character. The shell only executes the first line of a command field (the character string up to the percent sign). The shell interprets any other lines as standard input to the command that you specify.

For example, create a file called *mycron* and include the following *crontab* entry:

```
0 13 1,14 * * mailx -s "Call Mom!" $LOGNAME % now!
```

This *crontab* entry tells **cron** to execute **mailx.** The **mailx** command sends mail to the user specified by **$LOGNAME** and uses the **-s** option to set the Subject line. When **cron** encounters the percent character, it interprets it as a newline and passes the word "now!" to **mailx** which places it in the text of the mail message. The result is that, at 1:00 PM on the first and 14th day of every month, you get a reminder mail message with "Call Mom!" as the Subject, and "now!" as the message.

Submitting your crontab file

Once you create your *crontab* file, you must "submit" the file to the */usr/spool/cron/crontabs* directory where **cron** looks to execute commands. To do this, use the **crontab** command. The syntax for **crontab** looks like this:

crontab *file*

The **crontab** command copies the specified *file*, or standard input (the terminal) if no file is specified, into the *crontabs* directory that holds all the *crontab* files for the users on the system.

Now, whenever **cron** runs, it executes the commands in your *crontab* file.

To list the contents of your current *crontab* file, use the **-l** option to **cron.** You can remove your *crontab* file from the *crontabs* directory using **cron** with the **-r** option, for example:

```
cron -r mycron
```

For additional information on **cron**, see the **crontab**(C) manual page in the *User's Reference.*

Delaying program execution

The **batch** and **at** commands allow you to specify a command or sequence of commands to run at a later time. With the **batch** command, the system determines when to run the commands; with **at**, you determine when to run the commands.

Using batch

The **batch** command is useful for running a process or shell program that uses a large amount of system time. The **batch** command submits a batch job (a sequence of commands to execute) to the system; **batch** puts the job in a queue and runs it when the system load falls to an acceptable level. This frees the system to respond rapidly to other input and is a courtesy to other users. Note that if the system load is light, the system executes the submitted **batch** job immediately.

The general format for **batch** is as follows:

```
$ batch
first_command
.
.
.
last_command
$
```

When you have entered the last command, press ⟨Return⟩ followed by ⟨Ctrl⟩**d**.

The next example uses **batch** to execute the **grep** command at a convenient time. Here **grep** searches all files in the current directory for the string "dollar" and redirects the output to the file *dol.file*:

```
$ batch grep dollar * > dol.file
warning: commands will be executed using /bin/sh
job 681591549.b at Wed Aug  7  11:59:09 1991
$
```

When you submit a job with **batch**, the system responds with a job number, date, and time. This job number is not the same as the process number that the system generates when you run a command in the background.

To list your **batch** jobs, use the **-l** option to **batch**. For example:

```
batch -l
681591549.b     Wed Aug  7  11:59:09 1991
```

To cancel a **batch** job, use **-r** option. Use the job number to refer to the specific job. For example, to cancel the **batch** job in the example above, enter the following command:

```
batch -r 681591549.b
```

Using at

The **at** command allows you to specify an exact time to execute the commands. The syntax for **at** is as follows:

```
$ at time
first_command
 .
 .
 .
last_command
$
```

When you have entered the last command, press ⟨Return⟩ followed by ⟨Ctrl⟩**d**.

The *time* argument consists of between one and four digits, and AM or PM to show the time of day and the date. You do not need to enter a date if you want your job to run the same day. See the **at**(C) manual page in the *User's Reference* for other default times.

The following example shows how to use the **at** command to mail a "Happy Birthday" banner to the user *emily* on her birthday:

```
$ at 8:15am Feb 27
banner Happy Birthday | mail emily
⟨Ctrl⟩d
warning: commands will be executed using /bin/sh
job 453400603.a at Thurs Feb 27 08:15:00 1986
$
```

Notice that the **at** command, like **batch**, responds with the job number, date, and time.

If you decide that you do not want to execute the commands currently waiting in the **at** job queue, you can erase those jobs by using the **-r** option to **at**. The format for this is:

at -r ***jobnumber***

Try erasing the previous **at** job for the "Happy Birthday" banner using the following command:

at -r 453400603.a

If you have forgotten the job number, you can use the **-l** option to list the current jobs in the **at** queue. For example:

```
$ at -l
user = johnd 168302040.a at Thu Aug  1 16:56:55 1991
user = johnd 453400603.a at Fri Aug  2 08:15:00 1991
$
```

Notice that the system displays the job number and the time the job is scheduled to run.

Using the **at** command, mail yourself the file *memo* at noon to tell you it is lunch time:

```
$ at 12:00pm
mail lizp < memo
〈Ctrl〉d
job 263131754.a at Fri Aug  2 12:00:00 1991
$
```

Then, try listing the current **at** jobs using **at** with the **-l** option:

```
$ at -l
user = lizp 8263131754.a at Fri Aug 2 12:00:00 1989
$
```

Summary

On UNIX systems, you can run more than one command at a time. By running commands in the background, you can execute commands in the foreground without waiting for previous commands to finish. The system provides utilities for checking the status, terminating, and setting the priority of your processes. In addition, you can tell the system to run specified commands at a later time.

This chapter covers the following commands:

Table 5-2 Command review

Command	Description
ps(C)	checks the status of processes
nice(C)	runs processes at a lower priority
kill(C)	terminates a process
at(C),	runs a process at a specified time
batch(C),	delays the execution of a program until the system load is low
crontab(C)	executes programs repeatedly at a specified time

Chapter 6

Communicating with other sites

UNIX systems include a series of utilities that allow you to communicate with other computer sites. The particular utilities you use depend on how your computer is connected to the other site, what tasks you want to accomplish on the other site, and what operating system is running on the other site.

If the site is in close proximity to your computer, in the same room, for example, it is likely that the two computers are connected by a simple serial line. If the site is a UNIX site, use the UUCP commands discussed in "Using UUCP" below.

If the site you want to communicate with is remote (for example, on another floor, or across the country) then your computer is connected to it by telephone lines. If the site is a UNIX system or XENIX system site, use the UUCP commands discussed in "Using UUCP" below to transfer files between the two sites and execute commands on the remote site. If the site is not a UNIX system or XENIX system site, use the commands discussed in "Using cu" below.

The UUCP commands do not allow you to have an *interactive* session with the remote site. If you want to have an interactive session, use the commands discussed in "Using cu" below.

This chapter assumes that your UUCP network is configured already. If this is not true, refer to "Building a remote network with UUCP" in the *System Administrator's Guide* for more information.

Using UUCP

UUCP is a series of programs that provide networking capabilities for UNIX systems. While UUCP commands can be used over serial lines, they are usually used on computers connected by telephone lines.

The UUCP programs allow you to transfer files between remote computers and to execute commands on remote computers. Since the computers may be connected by telephone lines, UUCP transfers can take place over thousands of miles. A UUCP site in New York City can transfer a file to or execute a command on a connected UUCP site in San Francisco, or Jakarta, or anywhere in the world. The following sections explain how to use these UUCP programs.

Transferring files with uucp

Both the **uucp**(C) and **uuto**(C) commands can be used to transfer copies of binary and text files between remote UUCP sites. There are advantages and disadvantages to each. The **uucp** command gives you great flexibility in specifying where on the remote system the transferred file is to be placed. However, **uucp** syntax can be rather long and complicated. The **uuto** command, on the other hand, is easy to use. But **uuto** restricts where you can place the file on the remote system. In addition, retrieving a file sent with **uuto** is slightly more complicated than retrieving a file sent with **uucp**.

This section discusses the **uucp** command; the following section discusses the **uuto** command.

Before you begin

Before you can copy files to remote sites with **uucp**, you must verify that:

- Your local site is a "dial out" site.
- Your local site "knows" how to call the remote site.
- The files that you want to send have read permission set for others.
- The directory that contains the file that you want to send has read and execute permissions set for others. This allows the directory to be searched.
- Your computer has write permission in the directory on the remote site to which you want to copy the file.

Each of these is discussed below.

Some UUCP sites are "dial-in" sites, some are "dial-out" sites, and some are both. Verify that your site is a dial-out site. If it is not, your computer might have the capability to be on the receiving end of a UUCP connection, but not on the calling end.

You must be sure that your computer "talks" to the site with which you want to communicate. The **uuname**(C) command gives you this information. Entering **uuname** with no options lists the UUCP sites your computer talks to directly. Entering **uuname** with the **-l** option causes the name of your computer to be displayed.

Note that you might be able to communicate with a site that does not show up in a **uuname** listing. This is possible because UUCP sites are often "chained together." So if you know that a site you want to transfer files to communicates with a site that your system communicates with, you can send files to the first site through the second. An example is provided later under "Indirect transfers."

Finally, you must verify that your computer has write permission on the directory on the remote site to which you want to transfer files. Each remote UUCP site has a */usr/lib/uucp/Permissions* file. This file specifies the directories on that site from which your computer can read and to which your computer can write. You can only send a file to a directory on a remote site if your computer has write permissions on that directory, as specified on the remote site's */usr/lib/uucp/Permissions* file.

In order to copy a file to a remote UUCP site, the file must have read permission set for others and the directory that contains the file must have read and execute permissions set for others. Use the **l** command to examine the file's permissions and the **l -d** command to examine the directory's permissions. If the permissions are not correct, enter the following commands to set the correct permissions:

```
chmod o+r filename
chmod o+rx directory
```

By default, most UUCP sites permit calling-in computers to write to their */usr/spool/uucppublic* directory. Since there is no way to find out which directories your computer can write to on the remote site, short of contacting somebody at the site, the safest thing to do when making a UUCP transfer is to write to */usr/spool/uucppublic*. The procedure for doing this is outlined below.

Using uucp

The syntax of the **uucp** command is similar to the syntax of **cp**(C):

uucp [*options*] *src_computer!src_file dest_computer!dest_file*

These arguments mean the following:

src_file	The name of the file that you want to copy.
src_computer	The name of the computer on which *src_file* is located.
dest_file	The name of the copied file on the receiving computer. Usually, *src_file* and *dest_file* are the same.
dest_computer	The name of the computer on which *dest_file* is located.

There are several different ways to specify the location on the remote machine to which you want to transfer the file. The simplest is the *~/dest_file* specification. This is also the safest specification, because *~/dest_file* is expanded to */usr/spool/uucppublic/dest_file,* thereby assuring that the transfer will succeed.

For example, to send */usr/markt/transfile* on *machine1* to */usr/spool/uucppublic* on *machine2,* enter the following command:

```
uucp /usr/markt/transfile machine2!~/transfile
```

This command creates the file */usr/spool/uucppublic/transfile* on *machine2.*

If */usr/markt* is your current directory, you can copy *transfile* to *machine2* with the following command:

```
uucp transfile machine2!~/transfile
```

With the **uucp** command, files are not copied and sent immediately. Instead, copies are placed in a spool directory and sent once the appropriate daemon awakens. In the case of the UUCP programs, the daemon is the **uucico** daemon. Depending on how your system is configured, a **uucp** transfer might take place within minutes, or it might take hours.

NOTE As the exclamation mark has special meaning to the C shell, you must "escape" with a backslash (\) any exclamation marks that appear in a **uucp** command, if you are using the C shell. For a C-shell user, the command above is specified as:

```
uucp transfile machine2\!~/transfile
```

Another form of the command allows you to specify the full pathname of the copied file on the remote computer. This is for sending the file to a specific directory on the remote system. However, you must be sure that your computer has write permission on this directory, otherwise the transfer will fail.

As an example, suppose that you want to send *transfile* in */usr/markt* on *machine1* to the */usr/cindy* directory *machine2*. To do so, enter the following command:

```
uucp /usr/markt/transfile machine2!/usr/cindy/transfile
```

Note that the **uucp** command can be used to retrieve files from a remote site, in addition to copying files to a remote site. Using the example above, if your local computer is *machine2* and you want to send a copy of */usr/markt/transfile* on *machine1* to the */usr/cindy* directory on *machine2*, enter the following command:

```
uucp machine1!/usr/markt/transfile /usr/cindy/transfile
```

You can also use **~user** to specify a location on the remote computer. The **~user** argument is expanded to the pathname of the home directory of the person on the remote computer whose login is ***user***. For example, if */usr/cindy* is the home directory of a user whose login is *cindy* on *machine2*, enter the following command from the */usr/markt* directory on *machine1* to copy */usr/markt/transfile* to */usr/cindy*:

```
uucp transfile machine2!~cindy/transfile
```

The receiving computer expands *~cindy* to the full pathname of *cindy*'s home directory, creating */usr/cindy/transfile*. Again, your computer must have write permission in *cindy*'s home directory in order for this transfer to succeed.

Indirect transfers

You might be able to send files to a UUCP site not listed in a **uuname** listing. As an example, suppose that your local computer is connected to a UUCP site named *machine2*. Suppose also that *machine2* is connected to a UUCP site named *machine3*. You can send */tmp/transfile* on your local computer to */usr/spool/uucppublic* on *machine3*. To do this, specify the full UUCP address relative to your local computer:

```
uucp /tmp/transfile machine2!machine3!~/transfile
```

Note that each site name in the command line is followed by an exclamation mark. By placing several site names in a **uucp** command line, you can greatly extend the range of systems to which you can copy files with **uucp**. This is also true for the **uuto** and **uux**(C) commands discussed below.

uucp options

Several options are available for the **uucp** command. Some of the most useful are the **-m** and **-n** ***user*** options.

The **-m** option sends you mail reporting on the success or failure of the file transfer. The **-n** ***user*** option notifies the ***user*** on the machine to whom the files are sent of the file transfer.

Other options are available for use with **uucp**. Refer to **uucp**(C) for a complete list of these options.

Checking the status with uustat

You can use the **uustat**(C) command to check on the status of files you copied with **uucp**. To check on the status of all your **uucp** jobs, enter the following command:

uustat

Your output looks like the following:

```
1234 markt machine2 2/19-10:29 2/19-10:40 JOB IS QUEUED
```

Reading from left to right, the elements of this message are:

1234	This is the job number assigned to this **uucp** transfer.
markt	This is the user who requested the transfer.
machine2	This is the site name of the recipient's computer.
2/19-10:29	This is the date and time the job was queued in the spool directory.
2/19-10:40	This is the date and time of the **uustat** request.
JOB˜IS˜QUEUED	This "job status" message tells you the status of the job. In this case, JOB IS QUEUED tells you that the job is in the spool directory waiting to be sent. If the transfer is completed, **uustat** displays the message: COPY FINISHED, JOB DELETED.

Several options are available to use with **uustat**. Refer to **uustat**(C) for more information.

Transferring files with uuto

The **uuto** command allows you to copy files to the public directory of a UUCP site to which your system is connected. The public directory on most UNIX systems and XENIX systems is */usr/spool/uucppublic*. The syntax of **uuto** is:

uuto [*options*] *src_file dest_computer!login*

The ***login*** argument is the login of the user to whom you are sending files.

Before you can send a file with **uuto**, you must verify that:

- The file has read permission set for others.
- The directory that contains the file has read and execute permissions set for others.

If the permissions are not correct, enter the following commands to set the correct permissions:

> **chmod o+r** ***filename***
> **chmod o+rx** ***directory***

Files sent with **uuto** are placed in the directory:

> */usr/spool/uucppublic/receive/**login**/**src_computer***

In this example, ***login*** is the login of the user to whom you are sending files and ***src_computer*** is the site name of ***your*** system.

As an example, suppose that you want to send a copy of *transfile* in */tmp* on your computer, *machine1*, to a user whose login is *cindy* on *machine2*. To do so, enter the following command:

> **uuto /tmp/transfile machine2!cindy**

This command copies *transfile* to the following directory:

> *usr/spool/uucppublic/receive/cindy/machine1*

When the file transfer is complete, the recipient is notified by **mail** that the file has arrived. If the **-m** option is used on the **uuto** command line, the sender is notified by **mail** of the success or failure of the transfer.

Like **uucp**, files transferred with **uuto** are not transferred immediately after the command is entered. Instead, they are placed in a spool directory and sent when the **uucico** daemon awakens.

Retrieving files with uupick

In order to retrieve a file sent by **uuto**, you must use the **uupick**(C) command. To execute **uupick**, enter the following command:

> **uupick [-s system]**

The **uupick** program searches the public directory for any files sent to you. If it finds any, it responds with the following prompt:

```
from src_computer: file filename ?
```

The *src_computer* is the name of the sender's computer and *filename* is the name of the file transferred. In the example above, if the **uuto** transfer to *cindy* on *machine2* is successful, *cindy* sees the following **uupick** prompt:

```
from machine1: file transfile ?
```

Several options are available for responding to the **uupick** prompt. Two of the most useful are **m [*dir*]** and **d**. The **m [*dir*]** option tells **uupick** to move the file to directory ***dir***. Once in ***dir***, you can manipulate the file as you would any other file on your system. In the example above, *cindy* could enter the following in response to the **uupick** prompt:

m $HOME

This causes *transfile* to be moved from the public directory to *cindy*'s home directory. If no directory is specified after **m**, the file is moved to the recipient's current directory.

Entering **d** at the **uupick** prompt causes the file to be deleted from the public directory. You can quit **uupick** by entering **q**. Note other **uupick** options are available. Refer to **uupick**(C) for a complete list of these options.

Executing commands with uux

Use the **uux**(C) command to execute commands on remote UUCP sites and on files gathered from remote UUCP sites. For security reasons, the commands available for remote execution on a computer are often very limited. A computer's */usr/lib/uucp/Permissions* file lists the commands that can be executed remotely on that computer. If you attempt to execute a command not listed in this file, you receive mail indicating that the command cannot be executed on the computer in question.

The syntax of **uux** is:

uux [*options*] ***command_line***

The ***command_line*** argument looks like any other UNIX command line, with the exception that commands and filenames might be prefixed with *site-name!*.

The following is an example of how to execute a command on a remote system. The command causes */tmp/printfile* on *machine2* to be sent to *machine2's* default printer:

uux machine2!lp machine2!/tmp/printfile

Note that prefixing a site name to a command causes the command to be executed on that site.

The following is an example of how to execute a command on a local system on files gathered with **uux** from remote systems. Suppose that your local computer is connected to both *machine2* and *machine3*. Suppose also that you want to compare the contents of */tmp/chpt1* on *machine2* with */tmp/chpt1* on *machine3*. To do so, enter the following command:

```
uux "diff machine2!/tmp/chpt1 machine3!/tmp/chpt1 > diff.file"
```

This command compares the contents of the files on *machine2* and *machine3* and place the output in *diff.file* in the current directory on the local computer. Since there is no site name prefixed to the **diff** command, the command is executed locally.

Note that, in the example above, the **uux** command line is placed in quotation marks. This is because it contains the redirect symbol (>). In general, place the **uux** command line in quotation marks whenever the command line contains special shell characters such as " < ", " > ", " | ", and so forth.

Logging in to remote systems

The **ct**(C) command connects your system to a remote terminal with a modem attached. The **cu**(C) command connects your system to a remote system. The remote system can be attached via phone lines or via a simple serial line. These commands differ from the UUCP commands discussed above in that your session with the remote system is *interactive*. The remote system "sees" you as just another user on the system. Both **ct** and **cu** are discussed below.

Using ct

The **ct** command connects a local computer to a remote terminal equipped with a modem and allows a user on that terminal to log in to the computer. To do this, the command dials the phone number of the remote modem. The remote modem must be able to answer the call automatically. When **ct** detects that the call has been answered, it issues a **getty** (login) process for the remote terminal and allows a user on the terminal to log in on the computer.

This command is especially useful when issued from the opposite end, that is, from the remote terminal itself. If you are using a remote terminal and you want to avoid long distance charges, you can use **ct** to have the computer place a call to your terminal. To do so, simply call the computer, log in, and issue the **ct** command. The computer hangs up the line and calls your terminal back.

If **ct** cannot find an available dialer, it tells you that all dialers are busy and asks if it should wait until one becomes available. If you answer yes, it asks how long (in minutes) it should wait. If you answer no, **ct** quits.

The syntax of **ct** is:

ct *[options] telno*

The argument ***telno*** is the telephone number of the remote terminal.

As an example, suppose that you have a terminal with a modem attached at home and that you want to log in to the computer at work from this terminal. To avoid long distance charges, first call your work computer and log in. Then issue the **ct** command to make the computer hang up and call your terminal back. If your phone number is 932-3497, the **ct** command is:

ct -s 1200 9323497

The **-s** option tells **ct** to call the modem at 1200 baud. If no device is available on the computer at work, you see the following message after executing **ct**:

```
The one 1200 baud dialer is busy
Do you want to wait for dialer? (y for yes):
```

If you type **n** (no), the **ct** command exits. If you type **y** (yes), **ct** prompts you to specify how long **ct** should wait:

```
Time, in minutes?
```

If a dialer is available when you enter the **ct** command, you see the following message:

```
Allocated dialer at 1200 baud
```

This means that a dialer has been found. You are then asked if you want the line connecting your remote terminal to the computer to be dropped:

```
Proceed to hang-up? (y to hang-up, otherwise exit):
```

Since you want to avoid long-distance charges by having the computer call you, answer **y** (yes). You are then logged off and **ct** calls your remote terminal back.

As another example, suppose that you are logged in on a computer through a local terminal and that you want to connect a remote terminal to the computer. The phone number of the modem on the remote terminal is 932-3497. To connect the terminal, enter the following command:

nohup ct -h -s 1200 9323497 &

The **-h** option tells **ct** not to disconnect the local terminal (the terminal on which the command was issued) from the computer. After the command is executed, a login prompt is displayed on the remote terminal. The user can then log in and work on the computer just as on a local terminal.

Several options are available for **ct**. Refer to **ct**(C) for a complete list of these options.

Using cu

The **cu**(C) command connects your local computer to a remote computer and allows you to be logged in on both computers simultaneously. The remote computer does not have to be a UNIX system.

If the remote computer is a UNIX system, **cu** allows you to move back and forth between the two computers, transferring files and executing commands on both. Note that **cu** only allows you to transfer text files. You cannot transfer binary files with **cu**. To transfer binary files to a remote UNIX system, use **uucp**.

The syntax of the **cu** command is:

cu [*options*] *target*

The ***target*** argument can take one of three forms:

phone number	This is the number of the remote computer to which you want to connect. You can embed equal signs, which represent secondary dial tones, and dashes, which represent four-second delays, in the phone number. A sample phone number might be 4084551222--341. This number contains an area code and number, two dashes for an eight second delay and an extension.
system-name	This is the name of a system that is listed in the */usr/lib/uucp/Systems* file. The **cu** command obtains the telephone number and the baud rate of *system-name* from this file. The **-s**, **-n**, and **-l** options should not be used with *system-name*. To see the list of computers in the *Systems* file, enter **uuname**.
-l *line*	This is the device name of the serial line connected to the remote computer. It has the form *ttyXX*, where *XX* is the number of a serial line.
-l *line dir*	Connects directly with serial line instead of making a phone connection.

Several options are available with the **cu** command. Refer to **cu**(C) for a complete list of these options.

Once the connection is made, if the remote computer is a UNIX system, you are presented with a login prompt. Log in as you would if you were connected locally. When you finish working on the remote computer, log off as you would if you were connected locally. Then terminate the **cu** connection by entering a tilde followed by a period (~.). You are still logged in on the local computer.

As an example, suppose that you want to log in to a remote UNIX computer via the phone lines. Suppose also that the remote computer's number is 847-7867. To connect to the remote computer, enter the following command:

```
cu -s1200 8477867
```

The **-s1200** option causes **cu** to use a 1200 baud dialer. If the **-s** option is not specified, **cu** uses the first available dialer at the speed specified in the *Devices* file.

When the remote UNIX system answers the call, **cu** notifies you that the connection has been made by displaying the following message:

```
Connected
```

Next, you are prompted for your login:

```
login:
```

Enter your login and password. Once you enter this information, you can use this computer as if you were logged in locally. When you are finished, logout and then enter:

```
~.
```

This terminates the **cu** session.

cu command strings

Several "Command Strings" are available with **cu** that allow your local computer to communicate with a remote UNIX system. Two of the most useful are **take** and **put**.

The **take** command allows you to copy files from the remote computer to the local computer. Suppose, for example, that you want to copy a file named *proposal* in the current directory of the remote computer to your home directory on the local computer. To do so, enter the following command:

```
~%take proposal /tmp/proposal
```

Note that you have to prefix a tilde and a percent sign (~%) to the **take** command, and that the tilde must be placed at the start of a line. For this reason, it is a good idea to press ⟨Return⟩ before using **take**.

The **put** command allows you to do the opposite of **take**. It copies files from the local computer to the remote computer. Suppose, for example, that you want to copy a file named *minutes* from the */usr/spool/uucppublic* directory on the local computer to the */tmp* directory of the remote computer. Suppose also that you want the file to be called *minutes.9-18* on the remote computer. To do so, enter the following command:

```
~%put /usr/spool/uucppublic/minutes /tmp/minutes.9-18
```

Like the **take** command, you have to prefix a tilde and a percent sign (~%) to the **put** command, with the tilde coming at the beginning of a line. Note also that **take** and **put** copy only text files, and only to UNIX systems. You cannot use these commands to copy binary files.

NOTE The **cu** command cannot detect or correct transmission errors. After a file transfer, you can check for loss of data by running the **sum** command on both the file that was sent and the file that was received. This command reports the total number of bytes in each file. If the totals match, your transfer was probably successful. See the **sum**(C) manual page for details.

Other command strings are available for use with **cu**. For a complete list of these, see **cu**(C).

Summary

This chapter covers the following commands:

Table 6-1 Command review

Command	Description
uucp(C)	Transfers binary and text files between remote UUCP sites
uuto(C)	Copies files to the public directory of a UUCP site
uuname(C)	Lists the computers with which your computer communicates
chmod(C)	Changes file and directory permissions
uustat(C)	Checks the status of files you copied with **uucp**
uupick(C)	Retrieves files sent by **uuto**
uux(C)	Executes commands on remote UUCP sites and on files gathered from remote UUCP sites
ct(C)	Connects your system to a remote terminal with a modem attached.
cu(C)	Connects your system to a remote system.

Chapter 7
Using a secure system

Every computer system needs protection from people accessing the computer, disks and system files without the system administrator's permission. The operating system carries its own protection in the form of built-in security features not present in other UNIX systems. These features apply to all users of the system and are maintained by the system administrator.

This chapter describes security from the viewpoint of the ordinary user. If you find that your system does not use a feature discussed in this chapter, your administrator has switched it off in favor of standard non-trusted behavior.

This chapter includes the following:

- Terminology used to describe ways of enforcing and breaking security.
- The role of the security administrator.
- How to log into a trusted system, change password and use another account.
- How to issue commands on a trusted system.
- Recommended security practices and security tips.

Terminology

The following terms are used to describe ways of enforcing and breaking security:

A "Trojan Horse" is a program which masquerades as an innocent program. It allows a person to steal your data, corrupt your files, or gain access to your account.

A "login spoofing program" disguises itself as the login program in order to steal your password. The program displays the login prompt on the terminal and waits for you to type in your login name followed by password. When you respond to the prompts and enter your login name and password, the program stores the password and reports that your entry was incorrect. The spoofing program then ends and the correct login program starts.

A "protected subsystem" is a collection of files, devices, and commands which protects a set of resources or which performs security tasks.

The "trusted computing base" (TCB) of a system is the software, hardware and firmware that provide the system with security. The TCB of your equipment consists of the system hardware and firmware (supplied by the hardware vendor) and the operating system.

The security administrator

A security administrator is appointed to enforce security practices, monitor the system, trace attempts to breach security and return the system to a trusted state in the unlikely event of a security break-in.

Login security

This section describes how to log into a trusted system, change password and use another account. It also explains what to do if you have difficulty logging in.

Logging in

The following login prompts are displayed:

```
login: login name
Password: non-echoed password
```

When you enter your password correctly, the last times you successfully and (if applicable) unsuccessfully logged in are displayed:

```
Last successful login for login:date and time on ttyxxx
Last unsuccessful login for login:date and time on ttyxxx
```

If these times do not match your actions, consult your administrator immediately. Someone might have tried to log into your account.

What to do if you cannot log in

If you cannot log in, go through the following checklist:

- The security administrator has given your password a lifetime, which has now expired. Ask the administrator to change your password and re-open your account.
- The security administrator has set a limit to the number of unsuccessful login attempts you are allowed to make for your account. When you exceed this number your account is locked automatically. Ask the administrator to re-open the account. If you feel that you entered your login details correctly, tell the administrator immediately. It is possible that the system has been interfered with.
- The security administrator has set a limit to the number of unsuccessful login attempts allowed at your terminal. When this number is exceeded your account is locked automatically. Ask the administrator to re-open the account. If you believe that you entered your login details correctly, inform the administrator immediately.
- The security administrator has locked your account or terminal. To continue work you must ask the administrator to re-open the account.
- The security administrator has set a date by which your password expires. When your password expires you are prompted to change it.
- If you forget your password, ask the security administrator to change it.

Changing your password

The security administrator decides whether or not you can change password for yourself. The administrator can also set a minimum time period between changes of password.

If you are not allowed to change password for yourself

If you are not allowed to change password for yourself and you try to use the **passwd**(C) command, the following message appears:

```
Password cannot be changed.  Reason: Not allowed to
execute password for the given user.
```

In this case, you must ask the administrator to change your password.

If you are allowed to change password for yourself

If you are allowed to change the password for yourself, the administrator sets up your account to allow you to specify the password of your choice or have the system generate one for you.

When you use the **passwd**(C) command, you are prompted for your current password:

```
Old password:
```

When you type it in correctly the date and time of your last change of password are displayed:

```
Last successful password change for login: date and time
Last unsuccessful password change for login: date and time
```

Make sure that these messages reflect your last attempts to change password. If they do not, tell your administrator immediately.

The following prompt is then displayed:

```
                    Choose password

You can choose whether you pick your own password,
or have the system create one for you.

          1. Pick your own password
          2. Pronounceable password will be generated for you

Enter choice (default is 1):
```

If you enter **1**, you are prompted for your new password. You are then prompted to repeat your entry. Refer to the "Recommended security practices" section for guidelines on choosing a password.

If you select **2**, the system generates a password for you. The following message is displayed:

```
Generating random pronounceable password for login.
The password, along with the hyphenated version, is shown.
Hit (Return) or <ENTER> until you like the choice.
When you have chosen the password you want, type it in.
Note: Type your interrupt character or 'quit' to abort at any time.

Password:xxxxxx Hyphenation:xx-xx-xx  Enter password:
```

The generated password is displayed with a hyphenated version. The hyphenation separates the password into logical parts and is designed to help you commit the password to memory. Do not write the password down.

If you decide not to change your password type **quit** or your interrupt character (normally the ⟨Del⟩ key). Your last unsuccessful password change time is updated and the following message is displayed:

```
Password cannot be changed.  Reason: user stopped program.
```

Using another account

The **su**(C) command allows you to become another user without logging off. **su** cannot be used to simply assume the login of another user; instead, **su** can be used under four circumstances:

- The super user can "su" to any account.
- An administrative user with the **su** authorization can "su" to the super user account.
- A user can "su" to their own account.
- A system daemon can "su" to an account.

To use **su**, the appropriate password must be supplied (unless you are already a super user). If the password is correct, **su** executes a new shell with the effective user ID set to that of the specified user.

Using commands on a trusted system

The use of commands is restricted on a trusted system. You can issue certain commands only if the security administrator has given you the appropriate *authorization*. This section describes the different types of *authorization* and how they affect your use of commands.

Authorizations

The security mechanism has two types of authorization: kernel and subsystem. A kernel authorization allows you to run specific processes on the operating system. A subsystem authorization allows you to use the commands of a specific protected subsystem.

The kernel authorizations are as follows:

execsuid	This authorization allows you to run SUID programs. An SUID program gains access to all the files, processes and resources belonging to the person running the program or the owner of the program file.
chmodsugid	Allows you to change the setuid and setgid attributes of a file or directory, using the **chmod**(C) command.
chown	Allows you to change the ownership of files using the **chown**(C) command.
audittrail	Permits the use of the audit subsystem to monitor your own activities only. This can be useful for debugging of programs because a detailed record of system calls is generated by the autdit daemon. For more information, see "Using the audit subsystem" in the *System Administrator's Guide.*

There are two levels of subsystem authorization: primary and secondary. A primary subsystem authorization allows you to use the commands of a protected subsystem as an administrator. Primary authorizations are given to administrators and are fully described in the *System Administrator's Guide.* However, they can be given to ordinary users also. The primary authorizations are:

mem	This authorization allows you to use the **ps**(C) command to check the status of other users' processes, and the **ipcs**(C) command to report the status of inter-process communication. Without the authorization, you can only use these commands to report on processes belonging to you.
terminal	Allows you to use the **write**(C) command to communicate with other users. If you use the **write** command without the authorization, any control codes and escape sequences in your message are converted to ASCII characters.

A secondary subsystem authorization allows you to use the commands of a subsystem as an ordinary user (that is, you are not given administrative privilege). Secondary authorizations are described below:

printqueue	Allows you to view other users' jobs on the print queue.
printerstat	Allows you to use the **enable**(C) and **disable**(C) commands to change the status of printers.
queryspace	Allows you to use the **df**(C) command to query the amount of space available on the file systems.
su	Permits the use of the **su**(C) command to access another account (including *root*). Without this authorization, users can only access accounts they are defined as being responsible for.

The auths command

The **auths**(C) command allows you to list your kernel authorizations, and start up a shell so that you can issue commands with specific authorizations. The instructions below show you how to use the **auths** command with a variety of arguments. Examples are given for a user with execsuid and chown authorizations.

- The **auths** command without arguments lists your kernel authorizations. For example:

```
$ auths
Kernel authorizations: execsuid,chown
```

- The **auths** command with the **-a** option allows you to specify a restricted set of one or more of your authorizations. For example, the user with execsuid and chown authorizations can restrict themself to the chown authorization:

```
$ auths -a chown
$ auths
Kernel authorizations: chown
```

  To restore your authorizations, leave the shell started by the **auths -a** command.

- The **auths** command with the **-r** option allows you to specify which of your authorizations you wish to remove. For example:

```
$ auths -r chown
$ auths
Kernel authorizations: execsuid
```

  You must leave the shell started by the **auths -r** command to restore your authorizations.

- The **auths** command with the **-c** option allows you to issue a command instead of starting an interactive subshell. In the example below, chown authorization is removed and then the **auths**(C) command is issued. The result is a line listing the user's authorizations; the chown authorization is not included.

```
$ auths -r chown -c auths
Kernel authorizations: execsuid
```

When the user executes another list, the chown authorization is restored:

```
$ auths
Kernel authorizations: execsuid,chown
```

Security for files in sticky directories

A directory with the sticky bit set means that only the file owner and the super user can remove files from that directory. Other users are denied the right to remove files irrespective of directory permissions. Only the super user can place the sticky bit on a directory. Unlike with files, the sticky bit on directories remains there until the directory owner or super user removes the directory or applies **chmod**(C) or **chmod**(S) to it.

A sticky directory contains a "t" at the end of the permissions field, as in this example:

```
drwxrwxrwt   2 bin      bin      1088 Mar 18 21:10 tmp
```

Recommended security practices

This section gives a list of recommended security practices for the ordinary user.

Password security

It is your responsibility to protect your password. The careless use and maintenance of passwords represents the greatest threat to the security of a computer system. The security administrator can configure the system to be as restrictive or open as desired. Some of the guidelines listed in this section might be enforced by the system, including password length, complexity, and lifetime. The following are basic guidelines for choosing and maintaining passwords:

- A password should be at least eight characters in length and include letters, digits and punctuation marks. For example, **frAiJ6***.
- Do not use a password that is easy to guess. A password must not be a name, nickname, proper noun or word found in the dictionary. Do not use your birthdate or a number in your address.
- Do not use words spelled backwards.
- Do not start or end a password with a single number. For example, do not use **terry9** as your password.

- Use different passwords on different machines. Do not make the passwords reflect the names of the machines.
- Always keep your password secret. A password should never be written down, shared with another person, sent over electronic mail or spoken. Treat your password like the PIN number for your instant teller card.
- Never re-use a password. This just increases the probability of someone guessing it.
- Never type a password while someone is watching your fingers.

Good security habits

The following are some general guidelines for simple, good security habits:

- Remember to log out before leaving a terminal.
- Use the **lock**(C) utility when you leave your terminal, even for a short time. The **lock** command requests a password at the time of use, and then locks the terminal until the password is re-entered.
- Make certain that sensitive files are not publically readable.
- Keep any floppies or tapes containing confidential data (program source, database backups) under lock and key.
- If you notice strange files in your directories, or find other evidence that your account has been tampered with, tell your system administrator.

Logging in and out

The following guidelines include "things to look out for" as well as recommended practices for logging into your system and logging out.

- When logging into a trusted system, check that the reported last login and logout times are as you remember them. Look out for login attempts made when you are normally logged out of the system. Report any discrepancies to your security administrator immediately.
- Be careful how you type in your password.
- When you enter your password and the system reports an error, although you believe your entry to have been correct, tell your security administrator immediately. Check the reported last login time against the current time. If there is a discrepancy it is possible that a spoofing program (see "Terminology" section) has taken your password.

File security

Follow the guidelines below when you are creating, copying, and moving files. The list also includes security tips related to your startup scripts.

- When you create a file or directory your startup script determines the permissions given to the file or directory. Newly created files and directories should only be accessible by you (the owner) or the group. It is advisable to keep your files and directories secure in this way, by leaving your startup script set to the system default. If you wish to share a few files with other users, change the permissions on those files, individually.
- When you use the **cp**(C) command to copy an SUID file owned by someone else, the new file is also an SUID file and is owned by you. Note that when an SUID file is executed, it has access to all your files and directories. It is good practice to use the **chmod**(C) command to change the permissions on the file so that it can be accessed only by you.
- When you use the **cp** command to copy a file so as to create a new file, the new file takes the permissions of the original file. Remember to check the permissions of the new file and, if necessary, change them using the **chmod**(C) command.
- Remember that temporary directories are world-readable.
- Use the **ls**(C) command to check the permissions on your shell, mailer and startup files. If the files can be read and modified by other users, change the permissions using **chmod** so that only you have access to them.

Data encryption—commands and descriptions

If you have sensitive data that requires greater protection than that provided by access permission, you can encrypt the data. The encrypted file can not be read without a password. If somebody tries to read the encrypted file without a password, it cannot be understood.

NOTE You will only have data encryption capabilities if the **crypt**(C) software is installed on your system. This software is available only within the United States and must be requested from your distributor.

There are seven different commands used in data encryption. A brief summary of these commands appears in the following table.

Command Line	Description
crypt	This command is used to encode and decode files. The **crypt** command reads from the standard input or keyboard and writes to the standard output or terminal.
makekey	This command generates an encryption key.
ed -x	This command line edits a file that has already been encrypted, or creates a new encrypted file using the **ed** editor.
vi -x	This command line edits a file that has already been encrypted, or creates a new encrypted file using the vi editor.
ex -x	This command line edits a file that has already been encrypted, or creates a new encrypted file using the ex editor.
edit -x	This command line edits a file that has already been encrypted, or creates a new encrypted file using the **edit** editor.
X	This command encrypts a file while in the editor mode **ed**, **ex**, or **edit**).

crypt — encode/decode files

The **crypt** command encodes and decodes files for security. When using **crypt**, you have to assign a password (key) to encode the file. The same password is used to decode the file. You cannot read an encrypted file unless you use the correct password to decode it.

If you do not give a password with the **crypt** command, the system prompts you for one. For security, the screen does not display the password as you type it in.

Password security is the most vulnerable part of the **crypt** command. Anyone who figures out your password can look at your files. The best way to ensure your security is to select an uncommon group of characters. As with your login password, the password should be no more than eight letters or numbers long.

A file can be encrypted in the shell mode using **crypt**, or in the edit mode using the **-x** or **X** option. When you are ready to decrypt the file, you can use the **crypt** command in the shell mode. The following is the command format to encrypt a file:

```
crypt < oldfile > newfile
```

The system prompts you for a password.

Before removing the unencrypted ***oldfile***, make sure the encrypted ***newfile*** can be decrypted using the appropriate password. The ***oldfile*** is the file to be encrypted. The ***newfile*** is the name of the destination file for the encrypted text. Now, remove the ***oldfile***.

NOTE Always remember to remove the file (***oldfile***) from which you are encrypting because it is not encrypted. Only the *newfile* is encrypted.

Without any arguments, the **crypt** command takes standard input from the keyboard and encodes it before directing it to the standard output (the display). To encode an existing file, you must tell **crypt** to take its input (<) from a file instead of the keyboard. Similarly, you must tell **crypt** to send its output (>) to a new file instead of the display.

To decrypt a file, redirect the encrypted file to a new file you can read. The command to decrypt a file is as follows:

crypt < ***crypted_file*** > ***new_filename***

NOTE Always encrypt and decrypt files separately.

Encrypting and decrypting with editors

You can use the editors (**ed**, **edit**, **ex**, and **vi**) to either edit an existing file that has been encrypted or to create a new encrypted file by using the **-x** option. When encrypting a file, you have to assign a password to encode the file. The same password is used to decode the file. An encrypted file cannot be read unless the correct password is used to decode it.

Select an uncommon group of characters for the password. It should be no more than eight characters long.

The following is the command format for the editors (**ed**, **edit**, **ex**, and **vi**) using the -x option:

ed -x [***filename***]

edit -x [***filename***]

ex -x [***filename***]

vi -x [***filename***]

The **-x** option is used either to edit an existing file that has been encrypted or to create a new encrypted file. The ***filename*** variable is the name of the file that is being created or edited. The system prompts you for a password.

When you get ready to decrypt the file, you must use the **crypt** command from the shell.

The editor **X** command is another way to encrypt a file while in the editor mode. The **X** command only works with the **ed, edit,** or **ex** editors. (For the **vi** editor, type :X.) This command also needs a password to encrypt and decrypt files.

After you have edited the file, you can easily encrypt it again by using the **X** command as follows:

1. While still in the editor, enter **X** on a line by itself.
2. The system prompts you for a password.
3. Quit the file.

Summary

This chapter covers the following commands:

Table 7-1 Command review

Command	Description
su((C))	Allows you to use another account
auths (C)	allows you to list your kernel authorizations, and use commands with specific authorizations
crypt (C)	Encodes and decodes files for security
ed (C)	Edits or create an encrypted file
edit (C)	
ex (C)	
vi (C)	
passwd (C)	Allows you to change your password
chmod (C)	Changes file and directory permissions
chown (C)	Changes ownership of files
ps (C)	Checks the status of processes
write (C)	Communicates with other users
enable (C)	Changes the status of printers
disable (C)	
df (C)	Queries the amount of space available on the file systems
lock (C)	Locks your terminal

Chapter 8

The Bourne shell

When you log into a UNIX system, you communicate with one of several interpreters. This chapter discusses the shell command interpreter, **sh**. This interpreter is a UNIX program that supports a very powerful command language. Each invocation of this interpreter is called a shell; and each shell has one function: to read and execute commands from its standard input.

The shell gives the user a high-level language in which to communicate with the operating system, therefore you can perform tasks unheard of in less sophisticated operating systems. Commands that would normally have to be written in a traditional programming language can be written with just a few lines in a shell procedure. In other operating systems, commands are executed in strict sequence. With the shell, commands can be:

- combined to form new commands
- passed positional parameters
- added or renamed by the user
- executed within loops or executed conditionally
- created for local execution without fear of name conflict with other user commands
- executed in the background without interrupting a session at a terminal

Furthermore, commands can "redirect" command input from one source to another and redirect command output to a file, terminal, printer, or to another command. This provides flexibility in tailoring a task for a particular purpose.

Basic concepts

The shell itself (that is, the program that reads your commands when you log in or that is invoked with the **sh** command) is a program written in the C language; it is not part of the operating system proper, but an ordinary user program.

How shells are created

On a UNIX system, a process is an executing entity complete with instructions, data, input, and output. All processes have lives of their own, and might even start (or "fork") new processes. Thus, at any given moment several processes might be executing, some of which are "children" of other processes.

Users log into the operating system and are assigned a "shell" from which they execute. This shell is a personal copy of the shell command interpreter that is reading commands from the keyboard: in this context, the shell is simply another process.

In the UNIX system multitasking environment, files might be created in one phase and then sent off to be processed in the "background." This allows the user to continue working while programs are running.

Commands

The most common way of using the shell is by entering simple commands at your keyboard. A *simple command* is any sequence of arguments separated by spaces or tabs. The first argument (numbered zero) specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed as arguments to that command. For example, the following command line might be entered to request printing of the files *allan, barry,* and *calvin:*

```
lpr allan barry calvin
```

If the first argument of a command names a file that is *executable* (as indicated by an appropriate set of permission bits associated with that file) and is actually a compiled program, the shell, as parent, creates a child process that immediately executes that program. If the file is marked as being executable, but is not a compiled program, it is assumed to be a shell procedure, that is, a file of ordinary text containing shell command lines. In this case, the shell spawns another instance of itself (a *subshell*) to read the file and execute the commands inside it. From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so. This provides uniformity of invocation.

You can enter as many command lines as you want without waiting for the commands to complete their execution and for the prompt to reappear. This is because UNIX systems support character type-ahead. The system can hold up to 256 characters in the kernel buffers that read keyboard input.

How the shell finds commands

The shell normally searches for commands in three distinct locations in the file system. The shell attempts to use the command name as given; if this fails, it prepends the string */bin* to the name. If the latter is unsuccessful, it prepends */usr/bin* to the command name. The effect is to search, in order, the current directory, then the directory */bin,* and finally, */usr/bin.* For example, the **pr** and **man** commands are actually the files */bin/pr* and */usr/bin/man,* respectively. A more complex pathname can be given, either to locate a file relative to the user's current directory, or to access a command with an absolute pathname. If a given command name includes a slash (/) (for example, */bin/sort dir/cmd,*) the prepending is not performed. Instead, a single attempt is made to execute the command as named.

This mechanism gives the user a convenient way to execute public commands and commands in or near the current directory, as well as the ability to execute any accessible command, regardless of its location in the file structure. The current directory is usually searched first, therefore anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command does not affect a user who already has a private command with the same name. The particular sequence of directories searched can be changed by resetting the shell **PATH** variable. (Shell variables are discussed later in this chapter.)

Generation of argument lists

The arguments to commands are very often filenames. Sometimes, these filenames have similar, but not identical, names. To take advantage of this similarity in names, the shell lets the user specify patterns that match the filenames in a directory. If a pattern is matched by one or more filenames in a directory, then those filenames are automatically generated by the shell as arguments to the command.

Most characters in such a pattern match themselves, but there are also UNIX special characters that can be included in a pattern. These special characters are: the asterisk (*), which matches any string, including the null string; the question mark (?), which matches any one character; and any sequence of characters enclosed within brackets ([and]), which matches any one of the enclosed characters. Inside brackets, a pair of characters separated by a dash (-) matches any character within the range of that pair. Thus [a-de] is equivalent to [abcde].

Examples of metacharacter usage:

Metacharacter	Meaning
*	Matches all names in the current directory
temp	Matches all names containing *"temp"*
[a-f]*	Matches all names beginning with *"a"* through *"f"*
*.c	Matches all names ending in *".c"*
/usr/bin/?	Matches all single-character names in */usr/bin*

This pattern-matching capability saves typing and, more importantly, makes it possible to organize information in large collections of files that are named in a structured fashion, using common characters or extensions to identify related files.

Pattern matching has some restrictions. If the first character of a filename is a dot (.), it can be matched only by an argument that literally begins with a dot. If a pattern does not match any filenames, then the pattern itself is the result of the match.

Note that directory names should not contain any of the following characters:

```
* ? [ ]
```

If these characters are used, then infinite recursion might occur during pattern matching attempts.

Quoting mechanisms

Several characters, including "<, >, *, ?, [," and "]," have special meanings to the shell. To remove the special meaning of these characters requires some form of quoting. This is done by using single quotation marks (') or double quotation marks (") to surround a string. A backslash (\) before a single character provides this function. (Back quotation marks (') are used only for command substitution in the shell and do not hide the special meanings of any characters.)

All characters within single quotation marks are taken literally. Thus:

```
echostuff='echo $? $*; ls * | wc'
```

results in the string:

```
echo $? $*; ls  *  |  wc
```

being assigned to the variable **echostuff**, but it does *not* result in any other commands being executed.

Within double quotation marks, the special meaning of certain characters does persist, while all other characters are taken literally. The characters that retain their special meaning are the dollar sign ($), the backslash (\), the back quotation mark (`), and the double quotation mark (") itself. Thus, within double quotation marks, variables are expanded and command substitution takes place (both topics are discussed in later sections). However, any commands in a command substitution are unaffected by double quotation marks, so that characters such as asterisk (*) retain their special meaning.

To hide the special meaning of the dollar sign ($) and single and double quotation marks within double quotation marks, precede these characters with a backslash (\). Outside of double quotation marks, preceding a character with a backslash is equivalent to placing single quotation marks around that character. A backslash followed by a newline causes that newline to be ignored. The backslash-newline pair is therefore useful in allowing continuation of long command lines.

Some examples of quoting are displayed below:

Input	Shell interprets as:
´'´	the back quotation mark (`)
´"´	the double quotation mark (")
´`echo one`´	the one word "`echo one` "
"\""	the double quotation mark (")
"`echo one` "	the one word "one"
"` "	illegal (expects another `)
one two	the two words "one" & "two"
"one two"	the one word "one two"
´one two´	the one word "one two"
´one * two´	the one word "one * two"
"one * two"	the one word "one * two"
`echo one`	the one word "one"

Standard input and output

In general, most commands do not know or care whether their input or output is coming from or going to a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline. A few commands vary their actions depending on the nature of their input or output, either for efficiency, or to avoid useless actions (such as attempting random access I/O on a terminal or a pipe).

When a command begins execution, it usually expects that three files are already open: a "standard input," a "standard output," and a "diagnostic output" (also called "standard error"). A number called a *file descriptor* is associated with each of these files. By convention, file descriptor 0 is associated with the standard input, file descriptor 1 with the standard output, and file descriptor 2 with the diagnostic output. A child process normally inherits these files from its parent; all three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the terminal screen). The shell permits the files to be redirected elsewhere before control is passed to an invoked command.

An argument to the shell of the form "<*file*" or ">*file*" opens the specified file as the standard input or output (in the case of output, destroying the previous contents of *file*, if any). An argument of the form ">>*file*" directs the standard output to the end of *file*, thus providing a way to append data to the file without destroying its existing contents. In either of the two output cases, the shell creates *file* if it does not already exist. Thus, the following command alone on a line creates a zero-length file:

```
> output
```

The following appends to file *log* the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution; neither blank interpretation nor pattern matching of filenames occurs after these substitutions. This means that:

```
echo 'this is a test' > *.gal
```

produces a one-line file named **.gal*. Similarly, an error message is produced by the following command, unless you have a file with the name " ? ":

```
cat < ?
```

Special characters are *not* expanded in redirection arguments because redirection arguments are scanned by the shell *before* pattern recognition and expansion takes place.

Diagnostic and other outputs

Diagnostic output from UNIX system commands is normally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines.) You can redirect this error output to a file by immediately prepending the number of the file descriptor (2 in this case) to either output redirection symbol > or >>. The following line appends error messages from the **cc** command to the file named *ERRORS*:

```
cc testfile.c 2>> ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol *without* any intervening spaces or tabs; otherwise, the number will be passed as an argument to the command.

This method can be generalized to allow redirection of output associated with any of the first ten file descriptors (numbered 0-9). For instance, if *cmd* puts output on file descriptor 9, then the following line will direct that output to the file *savedata*:

```
cmd 9> savedata
```

A command often generates standard output and error output, and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose, for example, that *cmd* directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd >standard   2> error   9> data
```

Command lines and pipelines

A sequence of commands separated by the vertical bar (|) makes up a *pipeline*. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbors by *pipes*, that is, the output of each command (except the last one) becomes the input of the next command in line.

A *filter* is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read large amounts of data before producing output; **sort** is an example of the extreme case that requires all input to be read before any output is produced. The following is an example of a typical pipeline:

```
nroff -mm text | col | lpr
```

nroff is a text formatter available in the UNIX Text Processing System whose output might contain reverse line motions, **col** converts these motions to a form that can be printed on a terminal lacking reverse-motion capability, and **lpr** does the actual printing. The flag **-mm** indicates one of the commonly used formatting options, and ***text*** is the name of the file to be formatted.

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It might be helpful to try these at a terminal:

- **who**
  Prints the list of logged-in users on the terminal screen.
- **who >>log**
  Appends the list of logged-in users to the end of file *log*.
- **who | wc -l**
  Prints the number of logged-in users. (The argument to **wc** is pronounced "minus ell".)
- **who | pr**
  Prints a paginated list of logged-in users.
- **who | sort**
  Prints an alphabetical list of logged-in users.
- **who | grep bob**
  Prints the list of logged-in users whose login names contain the string *bob.*
- **who | grep bob | sort | pr**
  Prints an alphabetized, paginated list of logged-in users whose login names contain the string *bob.*
- **{ date; who | wc -l ; } >> log**
  Appends (to file *log*) the current date followed by the count of logged-in users. Be sure to place a space after the left brace and a semicolon before the right brace.
- **who | sed -e ´s/ .*//´ | sort | uniq -d**
  Prints only the login names of all users who are logged in more than once. Note the use of **sed** as a filter to remove characters trailing the login name from each line. (The ".*" in the **sed** command is preceded by a space.)

The **who** command does not *by itself* provide options to yield all these results — they are obtained by combining **who** with other commands. Note that **who** just serves as the data source in these examples. As an exercise, replace " who | " with " </etc/passwd " in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments can appear anywhere on the command line, even at the start. This means that:

```
< infile >outfile sort | pr
```

is the same as:

```
sort < infile > outfile | pr
```

Command substitution

Any command line can be placed within back quotation marks (`` `...` ``) so that the output of the command replaces the quoted command line itself. This concept is known as *command substitution.* The command or commands enclosed between back quotation marks are first executed by the shell and then their output replaces the whole expression, back quotation marks and all. This feature is used to assign the output of commands to shell variables. (Shell variables are described in the next section.)

For example:

```
today=`date`
```

assigns the string representing the current date to the variable "today"; for example "Tue Nov 26 16:01:09 EST 1985". The following command saves the number of logged-in users in the shell variable *users*:

```
users=`who | wc -l`
```

Any command that writes to the standard output can be enclosed in back quotation marks. Back quotation marks can be nested, but the inside sets must be escaped with backslashes (\). For example:

```
logmsg=`echo Your login directory is \`pwd\``
```

displays the line "your login directory is *name of login directory.*" Shell variables can also be given values indirectly by using the **read** and **line** commands. The **read** command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named.

For example:

```
read first init last
```

takes an input line of the form:

```
G. A. Snyder
```

and has the same effect as entering:

```
first=G. init=A. last=Snyder
```

The **read** command assigns any excess "words" to the last variable.

The **line** command reads a line of input from the standard input and then echoes it to the standard output.

Shell variables

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are referred to as *positional parameters*; these are the variables that are normally set only on the command line. Other shell variables are simply names to which the user or the shell itself can assign string values.

Positional parameters

When a shell procedure is invoked, the shell implicitly creates *positional parameters*. The name of the shell procedure itself in position zero on the command line is assigned to the positional parameter $0. The first command argument is called $1, and so on. The **shift** command can be used to access arguments in positions numbered higher than nine. For example, the following shell script might be used to cycle through command line switches and then process all succeeding files:

```
while test -n "$1"
        do case $1 in
               -a) A=aoption ; shift ;;
               -b) B=boption ; shift ;;
               -c) C=coption ; shift ;;
        -*) echo "bad option" ; exit 1 ;;
        *) process rest of files
        esac
done
```

You can explicitly force values into these positional parameters by using the **set** command. For example:

set abc def ghi

assigns the string "abc" to the first positional parameter, $1, the string "def" to $2, and the string "ghi" to $3. Note that $0 cannot be assigned a value in this way—it always refers to the name of the shell procedure; or in the login shell, to the name of the shell.

User-defined variables

The shell also recognizes alphanumeric variables to which string values can be assigned. A simple assignment has the syntax:

name=string

Thereafter, ***$name*** yields the value ***string***. A ***name*** is a sequence of letters, digits, and underscores that begins with a letter or an underscore. No spaces surround the equal sign (=) in an assignment statement. Note that positional parameters cannot appear on the left side of an assignment statement; they can only be set as described in the previous section.

More than one assignment can appear in an assignment statement, but beware: *the shell performs the assignments from right to left.* Thus, the following command line results in the variable "A" acquiring the value "abc":

A=$B B=abc

The following are examples of simple assignments. Double quotation marks around the right-hand side allow spaces, tabs, semicolons, and newlines to be included in a string, while also allowing variable substitution (also known as "parameter substitution") to occur. This means that references to positional parameters and other variable names that are prefixed by a dollar sign ($) are replaced by the corresponding values, if any. Single quotation marks inhibit variable substitution:

```
MAIL=/usr/mail/gas
echovar="echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

In the above example, the variable **echovar** has as its value the string consisting of the values of the first four positional parameters, separated by spaces, plus the string "echo". No quotation marks are needed around the string of asterisks being assigned to **stars** because pattern matching (expansion of asterisk, the question mark, and brackets) does not apply in this context. Note that the value of **$asterisks** is the literal string "$stars", *not* the string "*****", because the single quotation marks inhibit substitution.

In assignments, spaces are not re-interpreted after variable substitution, so that the following example results in **$first** and **$second** having the same value:

```
first='a string with embedded spaces'
second=$first
```

In accessing the values of variables, you can enclose the variable name in braces **{ ... }** to delimit the variable name from any following string. In particular, if the character immediately following the name is a letter, digit, or underscore, then the braces are required. For example, examine the following input:

```
a='This is a string'
echo "${a}ent test of variables."
```

Here, the **echo** command prints:

```
This is a stringent test of variables.
```

If no braces were used, the shell would substitute a null value for "$aent" and print:

```
test of variables.
```

The following variables are maintained by the shell. Some of them are set by the shell, and all of them can be reset by the user:

HOME	Initialized by the **login** program to the name of the user's *login directory,* that is, the directory that becomes the current directory upon completion of a login; **cd** without arguments switches to the *$HOME* directory. Using this variable helps keep full pathnames out of shell procedures. This is of great benefit when pathnames are changed, either to balance disk loads or to reflect administrative changes.
IFS	The variable that specifies which characters are *internal field separators*. These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set IFS to include that delimiter.) The shell initially sets **IFS** to include the blank, tab, and newline characters.
MAIL	The pathname of a file where your mail is deposited. If **MAIL** is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail each time you return to command level (e.g., by leaving the editor). **MAIL** is not set automatically; if desired, it should be set (and optionally "exported") in the user's *.profile*. (The **export** command and *.profile* file are discussed later in this chapter.) (The presence of mail in the standard mail file is also announced at login, regardless of whether **MAIL** is set.)
MAILCHECK	This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the **MAILPATH** or **MAIL** parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each prompt.
MAILPATH	A colon (:) separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by % and a message that will be printed when the modification time changes. The default message is *you have mail*.
SHACCT	If this parameter is set to the name of a file writable by the user, the shell will write an accounting record in the file for each shell procedure executed. Accounting routines such as **acctcom**(ADM) and **accton**(ADM) can be used to analyze the data collected.
SHELL	When the shell is invoked, it scans the environment for this name. If it is found and there is an 'r' in the file name part of its value, the shell becomes a restricted shell.

PATH	The variable that specifies the search path used by the shell in finding commands. Its value is an ordered list of directory pathnames separated by colons. The shell initializes **PATH** to the list *:/bin:/usr/bin* where a null argument appears in front of the first colon. A null anywhere in the path list represents the current directory. On some systems, a search of the current directory is *not* the default and the ***PATH*** variable is initialized instead to */bin:/usr/bin*. If you wish to search your current directory last, rather than first, use: **PATH=/bin:/usr/bin:** Below, the two colons together represent a colon followed by a null, followed by a colon, thus naming the current directory. You could possess a personal directory of commands (say, *$HOME/bin*) and cause it to be searched *before* the other three directories by using: **PATH=$HOME/bin::/bin:/usr/bin** **PATH** is normally set in your *.profile* file.
CDPATH	This variable defines the search path for the directory containing **arg**. Alternative directory names are separated by a colon (:). The default path is <null> (specifying the current directory). The current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If **arg** begins with a / then the search path is not used. Otherwise, each directory in the path is searched for **arg**.
PS1	The variable that specifies what string is to be used as the primary *prompt* string. If the shell is interactive, it prompts with the value of PS1 when it expects input. The default value of PS1 is "$ " (a dollar sign followed by a blank).
PS2	The variable that specifies the secondary prompt string. If the shell expects more input when it encounters a newline in its input, it prompts with the value of **PS2**. The default value for this variable is "> " (a greater-than symbol followed by a space).

In general, you should be sure to **export** all of the above variables so that their values are passed to all shells created from your login. Use **export** at the end of your *.profile* file. An example of an **export** statement follows:

```
export HOME IFS MAIL PATH PS1 PS2
```

Predefined special variables

Several variables have special meanings; the following are set *only* by the shell:

$# Records the number of arguments passed to the shell, not counting the name of the shell procedure itself. For instance, $# yields the number of the highest set positional parameter. Thus:

sh cmd a b c

automatically sets $# to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
        echo 'two or more args required'; exit
fi
```

$? Contains the exit status of the last command executed (also referred to as "return code," "exit code," or "value"). Its value is a decimal string. Most UNIX system commands return zero to indicate successful completion. The shell itself returns the current value of $? as its exit status.

$$ The process number of the current process. Because process numbers are unique among all existing processes, this string is often used to generate unique names for temporary files. The operating system provides no mechanism for the automatic creation and deletion of temporary files; a file exists until it is explicitly removed. Temporary files are generally undesirable objects; the UNIX system pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur.

The following example illustrates the recommended practice of creating temporary files; note that the directories */usr* and */usr/tmp* are cleared out if the system is rebooted.

```
#       use current process id
#       to form unique temp file
temp=/usr/tmp/$$
ls > $temp
#       commands here, some of which use $temp
rm -f $temp
#       clean up at end
```

$!	The process number of the last process run in the background (using the ampersand (&)). This is a string containing from one to five digits.

$-	A string consisting of names of execution flags currently turned on in the shell. For example, $- might have the value "xv" if you are tracing your output.

The shell state

The state of a given instance of the shell includes the values of positional parameters, user-defined variables, environment variables, modes of execution, and the current working directory.

The state of a shell can be altered in various ways. These include changing the working directory with the **cd** command, setting several flags, and by reading commands from the special file, *.profile,* in your login directory.

Changing directories

The **cd** command changes the current directory to the one specified as its argument. This can and should be used to change to a convenient place in the directory structure. Note that **cd** is often placed within parentheses to cause a subshell to change to a different directory and execute some commands without affecting the original shell.

For example, the first sequence below copies the file */etc/passwd* to */usr/you/passwd*; the second example first changes directory to */etc* and then copies the file:

```
cp /etc/passwd /usr/you/passwd
(cd /etc; cp passwd /usr/you/passwd)
```

Note the use of parentheses. Both command lines have the same effect.

If the shell is reading its commands from a terminal, and the specified directory does not exist (or some component cannot be searched), spelling correction is applied to each component of *directory,* in a search for the "correct" name. The shell then asks whether or not to try and change directory to the corrected directory name; an answer of *n* means "no", and anything else is taken as "yes."

The .profile file

The file named *.profile* is read each time you log in. It is normally used to execute special one-time-only commands and to set and export variables to all later shells. Only after commands are read and executed from *.profile,* does the shell read commands from the standard input — usually the terminal.

If you wish to reset the environment after making a change to the *.profile* file, enter:

```
. .profile
```

This command eliminates the need to log out and then log in again to execute *.profile.*

Execution flags

The **set** command lets you alter the behavior of the shell by setting certain shell flags. In particular, the **-x** and **-v** flags can be useful when invoking the shell as a command from the terminal. The flags **-x** and **-v** can be **set** by entering:

```
set -xv
```

The same flags can be turned *off* by entering:

```
set +xv
```

These two flags have the following meaning:

- -v Input lines are printed as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.
- -x Commands and their arguments are printed as they are executed. (Shell control commands, such as **for**, **while**, etc., are not printed, however.) Note that **-x** causes a trace of only those commands that are actually executed, whereas **-v** prints each line of input until a syntax error is detected.

The **set** command is also used to set these and other flags within shell procedures.

A command's environment

All variables and their associated values that are known to a command at the beginning of its execution make up its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the **export** command. The **export** command places the named variables in the environments of both the shell *and* all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line. Such variables are placed in the environment of the procedure being invoked. For example:

```
#      keycommand
echo $a $b
```

This is a simple procedure that echoes the values of two variables. If it is invoked as:

```
a=key1 b=key2 keycommand
```

then the resulting output is:

```
key1 key2
```

Keyword parameters are *not* counted as arguments to the procedure and do not affect $#.

A procedure can access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are not reflected in the environment; they are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made exportable from the current shell, enter:

```
export
```

You also get a list of variables that have been made **readonly**. To get a list of name-value pairs in the current environment, enter either:

```
printenv
```

or

```
env
```

Invoking the shell

The shell is a command and can be invoked in the same way as any other command:

sh *proc* [*arg* ...]	A new instance of the shell is explicitly invoked to read *proc*. Arguments, if any, can be manipulated.
sh -v *proc* [*arg* ...]	This is equivalent to putting "set -v" at the beginning of *proc*. It can be used in the same way for the **-x**, **-e**, **-u**, and **-n** flags.
proc [*arg* ...]	If *proc* is an executable file, and is not a compiled executable program, the effect is similar to that of: **sh proc args** An advantage of this form is that variables that have been exported in the shell will still be exported from *proc* when this form is used (because the shell only forks to read commands from *proc*). Thus any changes made within *proc* to the values of exported variables are passed on to subsequent commands invoked from *proc*.

Passing arguments to shell procedures

When a command line is scanned, any character sequence of the form *$n* is replaced by the *n*th argument to the shell, counting the name of the shell procedure itself as $0. This notation permits direct reference to the procedure name and to as many as nine positional parameters. Additional arguments can be processed using the **shift** command or by using a **for** loop.

The **shift** command shifts arguments to the left; i.e., the value of $1 is thrown away, $2 replaces $1, $3 replaces $2, and so on. The highest-numbered positional parameter becomes *unset* ($0 is never shifted). For example, in the shell procedure *ripple* below, **echo** writes its arguments to the standard output.

```
#       ripple command
while test $# != 0
do
        echo $1 $2 $3 $4 $5 $6 $7 $8 $9
        shift
done
```

Lines that begin with a number sign (#) are comments. The looping command, **while**, is discussed in "Conditional looping: while and until" in this chapter. If the procedure were invoked with:

```
ripple a b c
```

it would print:

```
a b c
b c
c
```

The special shell variable **star** ($*) causes substitution of all positional parameters except $0. Thus, the **echo** line in the *ripple* example above could be written more compactly as:

```
echo $*
```

These two **echo** commands are *not* equivalent: the first prints at most nine positional parameters; the second prints *all* of the current positional parameters. The shell star variable ($*) is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command. For example:

```
wc $*
```

counts the words of each of the files named on the command line.

It is important to understand the sequence of actions used by the shell in scanning command lines and substituting arguments. The shell first reads input up to a newline or semicolon, and then parses that much of the input. Variables are replaced by their values and then command substitution (via back quotation marks) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next, the shell scans the resulting command line for *internal field separators,* that is, for any characters specified by IFS to break the command line into distinct arguments; *explicit* null arguments (specified by "" or ´´) are retained, while implicit null arguments resulting from evaluation of variables that are null or not set are removed. Then filename generation occurs with all metacharacters being expanded. The resulting command line is then executed by the shell.

Sometimes, command lines are built inside a shell procedure. In this case, it is sometimes useful to have the shell rescan the command line after all the initial substitutions and expansions have been performed. The special command **eval** is available for this purpose. **eval** takes a command line as its argument and simply rescans the line, performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

```
command=who
output=´ | wc -l´
eval $command $output
```

This segment of code results in the execution of the command line:

```
who | wc -l
```

Uses of **eval** can be nested so that a command line can be evaluated several times.

Controlling the flow of control

The shell provides several commands that implement a variety of control structures useful in controlling the flow of control in shell procedures. Before describing these structures, a few terms need to be defined.

A *simple command* is any single irreducible command specified by the name of an executable file. I/O redirection arguments can appear in a simple command line and are passed to the shell, *not* to the command.

A *command* is a simple command or any of the shell control commands described below. A *pipeline* is a sequence of one or more commands separated by vertical bars (|). In a pipeline, the standard output of each command but the last is connected (by a *pipe*) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is the exit status of last process in the pipeline.

A *command list* is a sequence of one or more pipelines separated by a semicolon (;), an ampersand (&), an "and-if" symbol (&&), or an "or-if" (| |) symbol, and optionally terminated by a semicolon or an ampersand. A semicolon causes sequential execution of the previous pipeline. This means that the shell waits for the pipeline to finish before reading the next pipeline. On the other hand, the ampersand (&) causes asynchronous background execution of the preceding pipeline. Thus, both sequential and background execution are allowed. A background pipeline continues execution until it terminates voluntarily, or until its processes are killed.

Other uses of the ampersand include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you enter:

```
nohup cc prog.c&
```

You can continue working while the C compiler runs in the background. A command line ending with an ampersand is immune to interrupts or quits that you might generate by typing **INTERRUPT** or **QUIT**. However, ⟨Ctrl⟩d *will* abort the command if you are operating over a dial-up line or have *stty hupcl*. In this case, it is wise to make the command immune to hang-ups (i.e., logouts) as well. The **nohup** command is used for this purpose. In the above example without **nohup**, if you log out from a dial-up line while **cc** is still executing, **cc** will be killed and your output will disappear.

The ampersand operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of background processes without a compelling reason for doing so.

The **and-if** (&&) and **or-if** (| |) operators cause conditional execution of pipelines. Both of these are of equal precedence when evaluating command lines (but both are lower than the ampersand (&) and the vertical bar (|)). In the command line:

cmd1 | | cmd2

the first command, *cmd1,* is executed and its exit status examined. Only if *cmd1* fails (i.e., has a nonzero exit status) is *cmd2* executed. Thus, this is a more terse notation for:

```
if      cmd1
        test $? != 0
then
        cmd2
fi
```

The **and-if** operator (&&) yields a complementary test. For example, in the following command line:

cmd1 && cmd2

the second command is executed only if the first *succeeds* (and has a zero exit status). In the sequence below, each command is executed in order until one fails:

cmd1 && cmd2 && cmd3 && ... && cmdn

A simple command in a pipeline can be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line formats and prints two separate documents:

{ nroff -mm text1; nroff -mm text2; } | lpr

Note that a space is needed after the left brace and that a semicolon should appear before the right brace.

Using the if statement

The shell provides structured conditional capability with the **if** command. The simplest **if** command has the following form:

if ***command-list***
then ***command-list***
fi

The command list following the **if** is executed and if the last command in the list has a zero exit status, then the command list that follows **then** is executed.

The word **fi** indicates the end of the **if** command.

To cause an alternative set of commands to be executed when there is a nonzero exit status, an **else** clause can be given with the following structure:

> **if** ***command-list***
> **then** ***command-list***
> **else** ***command-list***
> **fi**

Multiple tests can be achieved in an **if** command by using the **elif** clause, although the **case** statement might be better for large numbers of tests. For example:

```
if      test -f "$1"
#                         is $1 a file?
then    pr $1
elif    test -d "$1"
#                         else, is $1 a directory?
then    (cd $1; pr *)
else    echo $1 is neither a file nor a directory
fi
```

The above example is executed as follows: if the value of the first positional parameter is a filename (-f), then print that file; if not, then check to see if it is the name of a directory (-d). If so, change to that directory (cd) and print all the files there (pr *). Otherwise, **echo** the error message.

The **if** command can be nested (but be sure to end each one with a **fi**). The newlines in the above examples of **if** can be replaced by semicolons.

The exit status of the **if** command is the exit status of the last command executed in any **then** clause or **else** clause. If no such command was executed, **if** returns a zero exit status.

Note that an alternate notation for the **test** command uses brackets to enclose the expression being tested. For example, the previous example might have been written as follows:

```
if      [ -f "$1" ]
#                         is $1 a file?
then    pr $1
elif    [ -d "$1" ]
#                         else, is $1 a directory?
then    (cd $1; pr *)
else    echo $1 is neither a file nor a directory
fi
```

Note that a space after the left bracket and one before the right bracket are essential in this form of the syntax.

Using the case statement

A multiple test conditional is provided by the **case** command. The basic format of the **case** statement is:

> **case** *string* **in**
>   ***pattern*** **)** ***command-list*** *;;*
>   **...**
>   ***pattern*** **)** ***command-list*** *;;*
> **esac**

The shell tries to match *string* against each pattern in turn, using the same pattern-matching conventions as in filename generation. If a match is found, the command list following the matched pattern is executed; the double semicolon (;;) serves as a break out of the **case** and is required after each command list except the last. Note that only one pattern is ever matched, and that matches are attempted in order, so that if a asterisk (*) is the first pattern in a **case**, no other patterns are looked at.

More than one pattern can be associated with a given command list by specifying alternate patterns separated by vertical bars (|).

```
case $i in
        *.c)    cc $i
                ;;
        *.h | *.sh)
                : do nothing
                ;;
        *)      echo "$i of unknown type"
                ;;
esac
```

In the above example, no action is taken for the second set of patterns because the null, colon (:) command is specified. The asterisk (*) is used as a default pattern, because it matches any word.

The exit status of **case** is the exit status of the last command executed in the **case** command. If no commands are executed, then **case** has a zero exit status.

Conditional looping: while and until

A **while** command has the general form:

> **while** ***command-list***
> **do**
>   ***command-list***
> **done**

The commands in the first ***command-list*** are executed, and if the exit status of the last command in that list is zero, then the commands in the second ***command-list*** are executed. This sequence is repeated as long as the exit status of the first ***command-list*** is zero. A loop can be executed as long as the first command-list returns a nonzero exit status by replacing **while** with **until.**

Any newline in the above example can be replaced by a semicolon. The exit status of a **while** (or **until**) command is the exit status of the last command executed in the second ***command-list.*** If no such command is executed, **while** (or **until**) has a zero exit status.

Looping over a list: for

Often, one wishes to perform some set of operations for each file in a set of files, or execute some command once for each of several arguments. The **for** command can be used to accomplish this. The **for** command has the format:

> **for** ***variable*** **in** ***word-list***
> **do**
> ***command-list***
> **done**

Here ***word-list*** is a list of strings separated by blanks. The commands in the ***command-list*** are executed once for each word in the ***word-list. variable*** takes on as its value each word from the word list, in turn. The word list is fixed after it is evaluated the first time. For example, the following **for** loop causes each of the C source files *xec.c, cmd.c,* and *word.c* in the current directory to be compared with a file of the same name in the directory */usr/src/cmd/sh*:

```
for CFILE in xec cmd word
do      diff $CFILE.c /usr/src/cmd/sh/$CFILE.c
done
```

Note that the first occurrence of **CFILE** immediately after the word **for** has no preceding dollar sign, since the name of the variable is wanted and not its value.

You can omit the "**in** *word-list*" part of a **for** command; this causes the current set of positional parameters to be used in place of word-list. This is useful when writing a command that performs the same set of commands for each of an unknown number of arguments.

As an example, create a file named *echo2* that contains the following shell script:

```
for word
do echo $word$word
done
```

Give *echo2* execute status:

```
chmod +x echo2
```

Now type the following command:

```
echo2 ma pa bo fi yo no so ta
```

The output from this command is:

```
mama
papa
bobo
fifi
yoyo
nono
soso
tata
```

Loop control: break and continue

The **break** command can be used to terminate execution of a **while** or a **for** loop. The **continue** command immediately starts the execution of the next iteration of the loop. These commands are effective only when they appear between **do** and **done**.

The **break** command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched **done**. Exit from *n* levels is obtained by **break** *n*.

The **continue** command causes execution to resume at the nearest enclosing **for, while,** or **until** statement, i.e., the one that begins the innermost loop containing the **continue.** You can also specify an argument *n* to **continue** and execution will resume at the *n*th enclosing loop:

```
# This procedure is interactive.
# "Break" and "continue" commands are used
# to allow the user to control data entry.
while true #loop forever
do      echo "Please enter data"
        read response
        case "$response" in
        "done") break
                # no more data
                ;;
        "")     # just a carriage return,
                # keep on going
                continue
                ;;
        *)      # process the data here
                ;;
        esac
done
```

End-of-file and exit

When the shell reaches the end-of-file in a shell procedure, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The top level shell is terminated by typing a ⟨Ctrl⟩**d** (which logs the user out of the system).

The **exit** command simulates an end-of-file, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated normally by placing "exit 0" at the end of the file.

Command grouping: parentheses and braces

There are two methods for grouping commands in the shell: parentheses and braces. Parentheses cause the shell to create a subshell that reads the enclosed commands. Both the right and left parentheses are recognized wherever they appear in a command line — they can appear as literal parentheses *only* when enclosed in quotation marks. For example, if you enter:

garble(stuff)

the shell prints an error message. Quoted lines, such as:

```
garble"("stuff")"
"garble(stuff)"
```

are interpreted correctly. Other quoting mechanisms are discussed in "Quoting mechanisms" in this chapter.

This capability of creating a subshell by grouping commands is useful when performing operations without affecting the values of variables in the current shell, or when temporarily changing the working directory and executing commands in the new directory without having to return to the current directory.

The current environment is passed to the subshell and variables that are exported in the current shell are also exported in the subshell. Thus:

```
CURRENTDIR=`pwd`; cd /usr/docs/otherdir;
nohup nroff doc.n > doc.out&; cd $CURRENTDIR
```

and

```
(cd /usr/docs/otherdir; nohup nroff doc.n > doc.out&)
```

accomplish the same result: */usr/docs/otherdir/doc.n* is processed by *nroff* and the output is saved in */usr/docs/otherdir/doc.out*. (Note that **nroff** is a text processing command.) However, the second example automatically puts you back in your original working directory. In the second example above, blanks or newlines surrounding the parentheses are allowed but not necessary. When entering a command line at your terminal, the shell will prompt with the value of the shell variable PS2 if an end parenthesis is expected.

Braces ({ and }) can also be used to group commands together. Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace can be followed by a newline (in which case the shell prompts for more input). Unlike parentheses, no subshell is created for braces: the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

Defining functions

The shell includes a function definition capability. Functions are like shell scripts or procedures except that they reside in memory and so are executed by the shell process, not by a separate process. The basic form is:

name ( ) {*list;*}

list can include any of the commands previously discussed. Functions can be defined in one section of a shell script to be called as many times as needed, making them easier to write and maintain. Here is an example of a function called "getyn":

```
# Prompt for yes or no answer - returns non-zero for no
getyn( )        {
        while   echo "$* (y/n)? \c" >& 2
        do      read yn rest
                case $yn in
                [yY])   return 0                        ;;
                [nN])   return 1                        ;;
                *)      echo "Please answer y or n" >&2 ;;
                esac
        done
}
```

In this example, the function appends a "(y/n)?" to the output and accepts "Y", "y", "n" or "N" as input, returning a 0 or 1. If the input is anything else, the function prompts the user for the correct input. (Echo should never fail, so the while-loop is effectively infinite.)

Functions are used just like other commands; an invocation of *getyn* might be:

getyn "Do you wish to continue" || exit

However, unlike other commands, the shell positional parameters **$1, $2, . . .,** are set to the arguments of the function. Since an exit in a function will terminate the shell procedure, the return command should be used to return a value back to the procedure.

Input/output redirection and control commands

The shell normally does *not* fork and create a new shell when it recognizes the control commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input to or output from each command. Also, when redirection of input or output is specified explicitly to a control command, a separate process is spawned to execute that command. Thus, when **if, while, until, case,** and **for** are used in a pipeline consisting of more than one command, the shell forks and a subshell runs the control command. This has two implications:

- Any changes made to variables within the control command are not effective once that control command finishes (this is similar to the effect of using parentheses to group commands).
- Control commands run slightly slower when redirected, because of the additional overhead of creating a shell for the control command.

Transfer between files: the dot command

A command line of the form:

```
. proc
```

causes the shell to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the dot command finishes. This is a good way to gather a number of shell variable initializations into one file. A common use of this command is to reinitialize the top level shell by reading the *.profile* file with:

```
. .profile
```

Interrupt handling: trap

Shell procedures can use the **trap** command to disable a signal (cause it to be ignored), or redefine its action. The form of the **trap** command is:

trap ***arg signal-list***

Here ***arg*** is a string to be interpreted as a command list and ***signal-list*** consists of one or more signal numbers as described in **signal**(S) in the *Programmer's Reference.* The most important of these signals follow:

Number	Signal
0	Exit from the shell
1	HANGUP
2	INTERRUPT character (DELETE or RUB OUT)
3	QUIT (⟨Ctrl⟩\)
9	KILL (cannot be caught or ignored)
11	Segmentation violation (cannot be caught or ignored)
15	Software termination signal

The commands in *arg* are scanned at least once, when the shell first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotation marks to surround these commands. The former inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the shell. The following procedure will print the name of the current directory in the user information as to how much of the job was done:

```
trap 'echo Directory was `pwd` when interrupted' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
        cd $i
           # commands to be executed in directory $i here
done
```

Beware that the same procedure with double rather than single quotation marks does something different. The following prints the name of the directory from which the procedure was first executed:

```
trap "echo Directory was `pwd` when interrupted" 2 3 15
```

A signal 11 can never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is interpreted by the **trap** command as a signal generated by exiting from a shell. This occurs either with an **exit** command, or by "falling through" to the end of a procedure. If *arg* is not specified, then the action taken upon receipt of any of the signals in the signal list is reset to the default system action. If *arg* is an explicit null string ('' or ""), then the signals in the signal list are ignored by the shell.

The **trap** command is most frequently used to make sure that temporary files are removed upon termination of a procedure. The preceding example would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm -f $temp; exit' 0 1 2 3 15
ls > $temp
     # commands that use $temp here
```

In this example, whenever signal 1 (hangup), 2 (interrupt), 3 (quit), or 15 (terminate) is received by the shell procedure, or whenever the shell procedure is about to exit, the commands enclosed between the single quotation marks are executed. The **exit** command must be included, or else the shell continues reading commands where it left off when the signal was received.

Sometimes the shell continues reading commands after executing trap commands. The following procedure takes each directory in the current directory, changes to that directory, prompts with its name, and executes commands typed at the terminal until an end-of-file (⟨Ctrl⟩d) or an interrupt is received. An end-of-file causes the **read** command to return a nonzero exit status, and thus the **while** loop terminates and the next directory cycle is initiated. An interrupt is ignored while executing the requested commands, but causes termination of the procedure when it is waiting for input:

```
d=`pwd`
for i in *
do      if test -d $d/$i
        then cd $d/$i
                while   echo "$i:"
                        trap exit 2
                        read x
                do      trap : 2
                        # ignore interrupts
                        eval $x
                done
        fi
done
```

Several **traps** can be in effect at the same time: if multiple signals are received simultaneously, they are serviced in numerically ascending order. To determine which traps are currently set, enter:

trap

It is important to understand some things about the way in which the shell implements the **trap** command. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing. When these (synchronous) processes terminate, normally or abnormally, the shell polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level. In this case, it is possible that no child process is running, so before the shell polls the traps, it waits for the termination of the first process spawned *after* the signal was received.

When a signal is redefined in a shell script, this does not redefine the signal for programs invoked by that script; the signal is merely passed along. A disabled signal is not passed.

For internal commands, the shell normally polls traps on completion of the command. An exception to this rule is made for the **read** command, for which traps are serviced immediately, so that **read** can be interrupted while waiting for input.

Shell script example

The following is a good shell script for handling signals.

```
:
#
#               Set signal handlers for shell script
#
trap "echo \" \nSignal caught   SIGHUP.DS IR drop \n \" " 1
trap "echo \" \nSignal caught   SIGINT interrupt DEL key \n \" "  2
trap "echo \" \nSignal caught   SIGQUIT Ctrl \\ \n \" ;rm core;exit 0 " 3
#
#       Note:  If you cd to a different directory you might want to
#       reset the trap for SIGQUIT so it will find the "core" file.
#       To do this you would put the same line below the cd command
#       in the shell script.
#
trap "echo \" \nSignal caught   SIGTERM  software termination\n \" "  15

echo " Going into loop "
while true
do
        cd /tmp
        trap "echo \" \nSignal caught SIGQUIT Ctrl \\ \n \";rm core;exit 0 " 3
        lf
        cd /usr
        trap "echo \" \nSignal caught SIGQUIT Ctrl \\ \n \";rm core;exit 0 " 3
        lf
        sleep 1
done
echo " Leaving the loop "
exit 0
```

Special shell commands

There are several special commands that are *internal* to the shell, some of which have already been mentioned. The shell does not fork to execute these commands, so no additional processes are spawned. These commands should be used whenever possible, because they are, in general, faster and more efficient than other UNIX system commands.

Several of the special commands have already been described because they affect the flow of control. They are dot (.), **break, continue, exit,** and **trap.** The **set** command is also a special command. Descriptions of the remaining special commands are given here:

:	The null command. This command does nothing and can be used to insert comments in shell procedures. Its exit status is zero (true). Its utility as a comment character has largely been supplanted by the number sign (#) which can be used to insert comments to the end-of-line. Beware: any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place, just as in other commands.
cd *arg*	Make *arg* the current directory. If *arg* is not a valid directory, or the user is not authorized to access it, a nonzero exit status is returned. Specifying **cd** with no *arg* is equivalent to entering "cd $HOME" which takes you to your home directory.
exec *arg ...*	If *arg* is a command, then the shell executes the command without forking and returning to the current shell. This is effectively a "goto" and no new process is created. Input and output redirection arguments are allowed on the command line. If *only* input and output redirection arguments appear, then the input and output of the shell itself are modified accordingly.
hash [-*r*] *name*	For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The **-r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented. *Hits* is the number of times a command has been invoked by the shell process. *Cost* is a measure of the work required to locate a command in the search path. There are certain

situations which require that the stored location of a command be recalculated. Commands for which this will be done are indicated by an asterisk (*) adjacent to the *hits* information. *Cost* will be incremented when the recalculation is done.

newgrp *arg* ...	The **newgrp** command is executed, replacing the shell. **Newgrp** in turn creates a new shell. Beware: only environment variables will be known in the shell created by the **newgrp** command. Any variables that were exported will no longer be marked as such.
pwd	Print the current working directory. See **pwd**(C) for usage and description.
read *var* ...	One line (up to a newline) is read from the standard input and the first word is assigned to the first variable, the second word to the second variable, and so on. All words left over are assigned to the *last* variable. The exit status of **read** is zero unless an end-of-file is read.
readonly *var* ...	The specified variables are made **readonly** so that no subsequent assignments can be made to them. If no arguments are given, a list of all **readonly** and of all exported variables is given.
return *n*	Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.
times	The accumulated user and system times for processes run from the current shell are printed.
type *name*	For each *name*, indicate how it would be interpreted if used as a command name.
ulimit [-f] *n*	This imposes a size limit of *n* blocks on files written. The **-f** flag imposes a size limit of *n* blocks on files written by child processes (files of any size can be read). With no argument, the current limit is printed. If no option is given and a number is specified, **-f** is assumed.

umask *nnn* — The user file creation mask is set to *nnn*. If *nnn* is omitted, then the current value of the mask is printed. This bit-mask is used to set the default permissions when creating files. For example, an octal umask of 137 corresponds to the following bit-mask and permission settings for a newly created file:

	user	group	other
Octal	1	3	7
bit-mask	001	011	111
permissions	rw-	r--	---

See **umask**(C) in the *User's Reference* for information on the value of *nnn*.

unset *name* — For each *name*, remove the corresponding variable or function. The variables **PATH, PS1, PS2, MAILCHECK** and **IFS** cannot be unset.

wait *n* — The shell waits for all currently active child processes to terminate. If *n* is specified, the shell waits for the specified process to terminate. The exit status of *wait* is always zero if *n* is not given; otherwise it is the exit status of child *n*.

Creation and organization of shell procedures

A shell procedure can be created in two simple steps. The first is building an ordinary text file. The second is changing the *mode* of the file to make it *executable,* thus permitting it to be invoked by:

```
proc args
```

rather than

```
sh proc args
```

The second step can be omitted for a procedure to be used once or twice and then discarded, but is recommended for frequently-used ones. For example, create a file named *mailall* with the following contents:

```
LETTER=$1
shift
for i in $*
do mail $i < $LETTER
done
```

Next enter:

```
chmod +x mailall
```

The new command might then be invoked from within the current directory by entering:

```
mailall letter joe bob
```

Here *letter* is the name of the file containing the message you want to send, and *joe* and *bob* are people you want to send the message to. Note that shell procedures must always be at least readable, so that the shell itself can read commands from the file.

If *mailall* were thus created in a directory whose name appears in the user's **PATH** variable, the user could change working directories and still invoke the *mailall* command.

Shell procedures are often used by users running the **csh**. However, if the first character of the procedure is a # (comment character), the **sh** assumes the procedure is a **csh** script, and invokes */bin/csh* to execute it. Always start **sh** procedures with some other character if **csh** users are to run the procedure at any time. This invokes the standard shell */bin/sh*.

Shell procedures can be created dynamically. A procedure can generate a file of commands, invoke another instance of the shell to execute that file, and then remove it. An alternate approach is that of using the *dot* command (.) to make the current shell read commands from the new file, allowing use of existing shell variables and avoiding the spawning of an additional process for another shell.

Many users prefer writing shell procedures to writing programs in C or other traditional languages. This is true for several reasons:

- A shell procedure is easy to create and maintain because it is only a file of ordinary text.
- A shell procedure has no corresponding object program that must be generated and maintained.
- A shell procedure is easy to create quickly, use a few times, and then remove.
- Shell procedures are usually short in length. They are written in a high-level programming language, are kept only in their source-language form, and are generally easy to find, understand, and modify.

By convention, directories that contain only commands and shell procedures are named *bin*. This name is derived from the word "binary", and is used because compiled and executable programs are often called "binaries" to distinguish them from program source files. Most groups of users sharing common interests have one or more *bin* directories set up to hold common procedures. Some users have their **PATH** variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard: it can become difficult to keep track of your environment and efficiency might suffer.

More about execution flags

There are several execution flags available in the shell that can be useful in shell procedures:

-e	This flag causes the shell to exit immediately if any command that it executes exits with a nonzero exit status. This flag is useful for shell procedures composed of simple command lines; it is not intended for use in conjunction with other conditional constructs.
-u	This flag causes unset variables to be considered errors when substituting variable values. This flag can be used to effect a global check on variables, rather than using conditional substitution to check each variable.
-t	This flag causes the shell to exit after reading and executing the commands on the remainder of the current input line. This flag is typically used by C programs which call the shell to execute a single command.
-n	This is a "don't execute" flag. On occasion, one might want to check a procedure for syntax errors, but not execute the commands in the procedure. Using "set -nv" at the beginning of a file will accomplish this.
-k	This flag causes all arguments of the form *variable=value* to be treated as keyword parameters. When this flag is *not* set, only such arguments that appear before the command name are treated as keyword parameters.

Supporting commands and features

Shell procedures can make use of any UNIX system command. The commands described in this section are either used especially frequently in shell procedures, or are explicitly designed for such use.

Conditional evaluation: test

The **test** command evaluates the expression specified by its arguments and, if the expression is true, **test** returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. **test** also returns a nonzero exit status if it has no arguments. Often it is convenient to use the **test** command as the first command in the command list following an **if** or a **while**. Shell variables used in **test** expressions should be enclosed in double quotation marks if there is any chance of their being null or not set.

The square brackets can be used as an alias to **test**, so that:

> [***expression***]

has the same effect as:

> **test** ***expression***

Note that the spaces before and after the ***expression*** in brackets are essential.

The following is a partial list of the options that can be used to construct a conditional expression:

-r ***file***	True if the named file exists and is readable by the user.
-w ***file***	True if the named file exists and is writable by the user.
-x ***file***	True if the named file exists and is executable by the user.
-s ***file***	True if the named file exists and has a size greater than zero.
-d ***file***	True if the named file is a directory.
-f ***file***	True if the named file is an ordinary file.
-z ***s1***	True if the length of string ***s1*** is zero.
-n ***s1***	True if the length of the string ***s1*** is nonzero.
-t ***fileds***	True if the open file whose file descriptor number is ***fildes*** is associated with a terminal device. If ***fildes*** is not specified, file descriptor 1 is used by default.
s1 = ***s2***	True if strings ***s1*** and ***s2*** are identical.
s1 != ***s2***	True if strings ***s1*** and ***s2*** are *not* identical.
s1	True if ***s1*** is *not* the null string.
n1 **-eq** ***n2***	True if the integers ***n1*** and ***n2*** are algebraically equal; other algebraic comparisons are indicated by **-ne** (not equal), **-gt** (greater than), **-ge** (greater than or equal to), **-lt** (less than), and **-le** (less than or equal to).

These can be combined with the following operators:

!	Unary negation operator.
-a	Binary logical AND operator.
-o	Binary logical OR operator; it has lower precedence than the logical AND operator (-a).
(*expr*)	Parentheses for grouping; they must be escaped to remove their significance to the shell. In the absence of parentheses, evaluation proceeds from left to right.

Note that all options, operators, filenames, etc. are separate arguments to **test**.

Echoing arguments

The **echo** command has the following syntax:

echo [*options*] [*args*]

echo copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a newline. You can use it to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic.

You can replace the **ls** command with

echo *

because the latter is faster and prints fewer lines of output.

The **-n** option to **echo** removes the newline from the end of the echoed line. Thus, the following two commands prompt for input and then allow entering on the same line as the prompt:

```
echo -n ´enter name:´
read name
```

The **echo** command also recognizes several escape sequences described in **echo**(C) in the *User's Reference.*

Expression evaluation: expr

The **expr** command provides arithmetic and logical operations on integers and some pattern-matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output; **expr** can be used inside grave accents to set a variable. Some typical examples follow:

```
#       increment $A
A=`expr $a + 1`
#       put third through last characters of
#       $1 into substring
substring=`expr "$1" : ´..\(.*\)´ `
#       obtain length of $1
c=`expr "$1" :  ´.* ´ `
```

The most common uses of **expr** are in counting iterations of a loop and in using its pattern-matching capability to pick apart strings.

True and false

The **true** and **false** commands perform the functions of exiting with zero and nonzero exit status, respectively. The **true** and **false** commands are often used to implement unconditional loops. For example, you might enter:

```
while true
do echo forever
done
```

This will echo "forever" on the screen until an INTERRUPT is entered.

In-line input documents

Upon seeing a command line of the form:

command << eofstring

where ***eofstring*** is any arbitrary string, the shell will take the subsequent lines as the standard input of ***command*** until a line is read consisting only of ***eofstring***. (By appending a minus (-) to the input redirection symbol (<<), leading tabs are deleted from each line of the input document before the shell passes the line to ***command***.)

The shell creates a temporary file containing the input document and performs variable and command substitution on its contents before passing it to the command. Pattern matching on filenames is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, you can quote any character of ***eofstring***:

```
command << \ eofstring
```

The in-line input document feature is especially useful for small amounts of input data, where it is more convenient to place the data in the shell procedure than to keep it in a separate file. For instance, you could enter:

```
cat <<- xx
        This message will be printed on the
        terminal with leading tabs removed.
xx
```

This in-line input document feature is most useful in shell procedures. Note that in-line input documents may not appear within grave accents.

Input/output redirection using file descriptors

We mentioned above that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with any file descriptor by using the **write**(S) system call (see the *Programmer's Reference*). The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By entering:

fd1 >& *fd2*

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* to the file associated with *fd2*. The default value for *fd1* and *fd2* is 1. If, at run time, no file is associated with *fd2*, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by entering:

```
command 2>&1
```

If you wanted to redirect both standard output and standard error output to the same file, you would enter:

```
command 1>file 2>&1
```

The order here is significant: first, file descriptor 1 is associated with *file*; then file descriptor 2 is associated with the same file as is currently associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file*, because at the time of the error output redirection, file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard input. You could enter:

```
fda <& fdb
```

to cause both file descriptors *fda* and *fdb* to be associated with the same input file. If *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for a command that uses two or more input sources.

Conditional substitution

Normally, the shell replaces occurrences of ***$variable*** by the string value assigned to ***variable***, if any. However, there exists a special notation to allow conditional substitution, dependent upon whether the variable is set or not null. By definition, a variable is set if it has ever been assigned a value. The value of a variable can be the null string, which can be assigned to a variable in anyone of the following ways:

```
A=
bcd=""
efg=''
set ''   ""
```

The first three examples assign null to each of the corresponding shell variables. The last example sets the first and second positional parameters to null. The following conditional expressions depend upon whether a variable is set and not null. Note that the meaning of braces in these expressions differs from their meaning when used in grouping shell commands. ***parameter*** as used below refers to either a digit or a variable name.

${***variable***:-***string***}	If ***variable*** is set and is nonnull, then substitute the value ***$variable*** in place of this expression. Otherwise, replace the expression with ***string***. Note that the value of ***variable*** is *not* changed by the evaluation of this expression.
${***variable***:=***string***}	If ***variable*** is set and is nonnull, then substitute the value $***variable*** in place of this expression. Otherwise, set ***variable*** to ***string***, and then substitute the value ***$variable*** in place of this expression. Positional parameters cannot be assigned values in this fashion.
${***variable***:?***string***}	If ***variable*** is set and is nonnull, then substitute the value of ***variable*** for the expression. Otherwise, print a message of the form ***variable***: ***string*** and exit from the current shell. (If the shell is the login shell, it is not exited.) If ***string*** is omitted in this form, then the message ***variable***: parameter null or not set is printed instead.
${***variable***:+***string***}	If ***variable*** is set and is nonnull, then substitute ***string*** for this expression. Otherwise, substitute the null string. Note that the value of ***variable*** is not altered by the evaluation of this expression.

These expressions can also be used without the colon. In this variation, the shell does not check whether the variable is null or not; it only checks whether the variable has ever been set.

The two examples below illustrate the use of this facility:

1. This example performs an explicit assignment to the **PATH** variable:

   PATH=${PATH:-':/bin:/usr/bin'}

   This says, if **PATH** has ever been set and is not null, then it keeps its current value; otherwise, set it to the string ":/bin:/usr/bin".

2. This example automatically assigns the **HOME** variable a value:

   cd ${HOME:='/usr/gas'}

   If **HOME** is set, and is not null, then change directory to it. Otherwise set **HOME** to the given value and change directory to it.

Invocation flags

There are five flags that can be specified on the command line when invoking the shell. These flags cannot be turned on with the **set** command:

-i If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is *interactive*. In such a shell, INTERRUPT (signal 2) is caught and ignored, and TERMINATE (signal 15) and QUIT (signal 3) are ignored.

-s If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. All remaining arguments specify the positional parameters.

-c When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored.

-t When this flag is on, a single command is read and executed, then the shell exits. This flag is not useful interactively, but is intended for use with C programs.

-r If this flag is present the shell is a restricted shell (see **rsh**(C)).

Effective and efficient shell programming

This section outlines strategies for writing efficient shell procedures, ones that do not waste resources in accomplishing their purposes. The primary reason for choosing a shell procedure to perform a specific function is to achieve a desired result at a minimum human cost. Emphasis should always be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, you should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume an inordinate amount of a system's resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. The user should become familiar with the **time** command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by variations in system load.

Number of processes generated

When large numbers of short commands are executed, the actual execution time of the commands might be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping, and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is the alphabetical list of those that are built in:

break	case	cd	continue	echo
eval	exec	exit	export	for
if	read	readonly	return	set
shift	test	times	trap	umask
until	wait	while	.	:
{}				

Parentheses, ( ), are built into the shell, but commands enclosed within them are executed as a child process, i.e., the shell does a **fork**, but no **exec**. Any command not in the above list requires both **fork** and **exec**.

The user should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes created (not necessarily simultaneously) can be described by:

processes = (k*n) + c

where *k* and *c* are constants for any given script, and *n* can be the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero.

Any procedure whose complexity measure includes n^2 terms or higher powers of n is likely to be intolerably expensive.

As an example, here is an analysis of a procedure named *split,* whose text is given below:

```
:
#       split
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
cat > temp$$
                # read stdin into temp file
                # save original lengths of $1, $2
if test -s "$1"
then start1=`wc -l < $1`
fi
if test -s "$2"
then start2=`wc -l < $2`
fi
grep "$b" temp$$ >> $1
                # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
                # lines without letters onto $2
total=" `wc-l< temp$$` "
end1=" `wc-l<$1` "
end2=" `wc-l<$2` "
lost=" `expr $total - \($end1 - $start1\) \
- \($end2 - $start2\)` "
echo "$total read, $lost thrown away"
```

For each iteration of the loop, there is one **expr** plus either an **echo** or another **expr**. One additional **echo** is executed at the end. If n is the number of lines of input, the number of processes is (2*n)+1.

Some types of procedures should *not* be written using the shell. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C. Shell procedures should not be used to scan or build files a character at a time.

Number of data bytes accessed

It is worthwhile to consider any action that reduces the number of bytes read or written. This might be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when the order is irrelevant. For instance, the second of the following examples is likely to be faster because the input to **sort** will be much smaller:

```
sort file | grep pattern
grep pattern file | sort
```

Shortening directory searches

Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long pathnames. Judicious use of **cd**, the *change directory* command, can help shorten long pathnames and thus reduce the number of directory searches needed. As an exercise, try the following commands:

```
ls -l /usr/bin/* >/dev/null
cd /usr/bin; ls -l * >/dev/null
```

The second command runs faster because of the fewer directory searches.

Directory-search order and the PATH variable

The **PATH** variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result might be a great increase in system overhead.

The process of finding a command involves reading every directory included in every pathname that precedes the needed pathname in the current **PATH** variable. As an example, consider the effect of invoking **nroff** (i.e., */usr/bin/nroff*) when the value of **PATH** is ":/bin:/usr/bin". The sequence of directories read is:

```
.
/
/bin
/
/usr
/usr/bin
```

A long path list assigned to **PATH** can increase this number significantly.

The vast majority of command executions are of commands found in */bin* and, to a somewhat lesser extent, in */usr/bin.* Careless **PATH** setup can lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best with respect to the efficiency of command searches:

:/usr/john/bin:/usr/localbin:/bin:/usr/bin
:/bin:/usr/john/bin:/usr/localbin:/usr/bin
:/bin:/usr/bin:/usr/john/bin:/usr/localbin
:/bin:/usr/bin:/usr/john/bin:/usr/localbin

The first one above should be avoided. The others are acceptable and the choice among them is dictated by the rate of change in the set of commands kept in */bin* and */usr/bin.*

A procedure that is expensive because it invokes many short-lived commands can often be speeded up by setting the **PATH** variable inside the procedure so that the fewest possible directories are searched in an optimum order.

Good ways to set up directories

It is wise to avoid directories that are larger than necessary. You should be aware of several special sizes. A directory that contains entries for up to 62* files (plus the required . and ..) fits in a single disk block and can be searched very efficiently. A directory can have up to 638* entries and still be viable, as long as it is used only for data storage; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink. This is very important to understand, because if your directory ever exceeds either the 62 or 638 thresholds, searches will be inefficient; furthermore, even if you delete files so that the number of files is less than either threshold, the system will still continue to treat the directory inefficiently.

Shell procedure examples

The power of the UNIX system shell command language is most readily seen by examining how many labor-saving UNIX system utilities can be combined to perform powerful and useful commands with very little programming effort. This section gives examples of procedures that do just that. By studying these examples, you will gain insight into the techniques and shortcuts that can be used in programming shell procedures (also called "scripts"). Note the use of the null command (:) to begin each shell procedure and the use of the number sign (#) to introduce comments.

* These figures apply to filenames of 14 characters or less. As filename lengths increase, up to a maximum of 255 characters, the number of files that fit on a single disk block decreases, thus reducing the optimum number of files in a directory.

It is intended that the following steps be carried out for each procedure:

1. Place the procedure in a file with the indicated name.
2. Give the file execute permission with the **chmod** command.
3. Move the file to a directory in which commands are kept, such as your own *bin* directory.
4. Make sure that the path of the *bin* directory is specified in the **PATH** variable found in *.profile.*
5. Execute the named command.

BINUNIQ

```
:
ls /bin /usr/bin | sort | uniq -d
```

This procedure determines which files are in both */bin* and */usr/bin.* It is done because files in */bin* will "override" those in */usr/bin* during most searches and duplicates need to be weeded out. If the */usr/bin* file is obsolete, then space is being wasted; if the */bin* file is outdated by a corresponding entry in */usr/bin* then the wrong version is being run and, again, space is being wasted. This is also a good demonstration of "sort | uniq" to find matches and duplications.

COPYPAIRS

```
:
#       Usage: copypairs file1 file2 ...
#       Copies file1 to file2, file3 to file4, ...
while test "$2" != ""
do
        cp $1 $2
        shift; shift
done
if test "$1" != ""
        then echo "$0: odd number of arguments" >&2
fi
```

This procedure illustrates the use of a **while** loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop, because you can adjust the positional parameters with the **shift** command to handle related arguments.

COPYTO

```
:
#       Usage: copyto dir file . . .
#       Copies argument files to "dir",
#       making sure that at least
#       two arguments exist, that "dir" is a directory,
#       and that each additional argument
#       is a readable file.
if test $# -lt 2
        then    echo "$0: usage: copyto directory file . . .">&2
elif test ! -d $1
        then    echo "$0: $1 is not a directory";>&2
else    dir=$1; shift
        for eachfile
        do      cp $eachfile $dir
        done
fi
```

This procedure uses an **if** command with several parts to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first; the original **$1** is shifted off.

DISTINCT1

```
:
#       Usage: distinct1
#       Reads standard input and reports list of
#       alphanumeric strings that differ only in case,
#       giving lowercase form of each.
tr -cs 'A-Za-z0-9' '\012' | sort -u | \
tr 'A-Z' 'a-z' | sort | uniq -d
```

This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. Note the use of the backslash at the end of the first line as the line continuation character. It might not be immediately obvious how this command works. You might wish to consult **tr**(C), **sort**(C), and **uniq**(C) in the *User's Reference* if you are completely unfamiliar with these commands. The **tr** command translates all characters except letters and digits into newline characters, and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The "uniq -d" prints (once) only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline relies on the fact that pipes and files can usually be interchanged. The first line below is equivalent to the last two lines, assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3

cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3
rm temp[123]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output.

Although pipelines can give a concise notation for complex processes, you should exercise some restraint, since such practice often yields incomprehensible code.

DRAFT

```
:
#       Usage: draft file(s)
#       Print manual pages for Diablo printer.
for i in $*
        do nroff -man $i | lpr
done
```

Users often write this kind of procedure for convenience in dealing with commands that require the use of distinct flags that cannot be given default values that are reasonable for all (or even most) users.

EDFIND

```
:
#       Usage: edfind file arg
#       Finds the last occurrence in "file" of a line
#       whose beginning matches "arg", then prints
#       3 lines (the one before, the line itself,
#       and the one after)
ed - $1 << -EOF
        ?^$2?
        -,+p
        q
EOF
```

This illustrates the practice of using **ed** in-line input scripts into which the shell can substitute the values of variables.

EDLAST

```
:
#       Usage: edlast file
#       Prints the last line of file,
#       then deletes that line.
ed - $1 <<-\!
        $p
        $d
        w
        q
!
echo done
```

This procedure illustrates taking input from within the file itself up to the exclamation point (!). Variable substitution is prohibited within the input text because of the backslash.

FSPLIT

```
:
#       Usage: fsplit file1 file2
#       Reads standard input and divides it into 3 parts
#       by appending any line containing at least one letter
#       to file1, appending any line containing digits but
#       no letters to file2, and by throwing the rest away.
count=0 gone=0
while read next
do
        count="`expr $count + 1`"
        case "$next"  in
        *[A-Za-z]*)
                echo  "$next"  >> $1 ;;
        *[0-9]*)
                echo  "$next"  >> $2 ;;
        *)
                gone="`expr $gone + 1` "
        esac
done
echo "$count lines read, $gone thrown away"
```

Each iteration of the loop reads a line from the input and analyzes it. The loop terminates only when **read** encounters an end-of-file. Note the use of the **expr** command.

Do not use the shell to read a line at a time unless you must because it can be an extremely slow process.

LISTFIELDS

```
:
grep $* | tr ":" "\012"
```

This procedure lists lines containing any desired entry that is given to it as an argument. It places any field that begins with a colon on a newline. Thus, if given the following input:

joe newman: 13509 NE 78th St: Redmond, Wa 98062

listfields produces this:

```
joe newman
13509 NE 78th St
Redmond, Wa 98062
```

Note the use of the **tr** command to transpose colons to linefeeds.

MKFILES

```
:
#       Usage: mkfiles pref [quantity]
#       Makes "quantity" files, named pref1, pref2, ...
#       Default is 5 as determined on following line.
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
        > $1$i
        i="`expr $i + 1`"
done
```

The *mkfiles* procedure uses output redirection to create zero-length files. The **expr** command is used for counting iterations of the **while** loop.

NULL

```
:
#       Usage: null files
#       Create each of the named files as an empty file.
for eachfile
do
        >$eachfile
done
```

This procedure uses the fact that output redirection creates the (empty) output file if a file does not already exist.

PHONE

```
:
#       Usage: phone initials ...
#       Prints the phone numbers of the
#       people with the given initials.
echo ′inits    ext     home′
grep "$1" <<END
        jfk     1234    999-2345
        lbj     2234    583-2245
        hst     3342    988-1010
        jqa     4567    555-1234
END
```

This procedure is an example of using an in-line input script to maintain a small database.

TEXTFILE

```
:
if test "$1" = "-s"
then
#       Return condition code
        shift
        if test -z "`$0 $*`" # check return value
        then
                exit 1
        else
                exit 0
        fi
fi

if test $# -lt 1
then    echo "$0: Usage: $0 [ -s ] file ..." 1>&2
        exit 0
fi

file $* | fgrep ′ text′ | sed ′s/:     .*//′
```

To determine which files in a directory contain only textual information, *textfile* filters argument lists to other commands. For example, the following command line will print all the text files in the current directory:

pr `textfile *` | lpr

This procedure also uses an **-s** flag which silently tests whether any of the files in the argument list is a text file.

WRITEMAIL

```
:
#       Usage: writemail message user
#       If user is logged in,
#       writes message to terminal;
#       otherwise, mails it to user.
echo "$1" | { write "$2" || mail "$2" ;}
```

This procedure illustrates the use of command grouping. The message specified by $1 is piped to both the **write** command and, if **write** fails, to the **mail** command.

Shell grammar

item:	***word***
	input-output
	name = value
simple-command:	***item***
	simple-command item
command:	***simple-command***
	(*command-list*)
	{ *command-list* }
	for *name* do *command-list* done
	for *name* in *word* do *command-list* done
	while *command-list* do *command-list* done
	until *command-list* do *command-list* done
	case *word* in *case-part* esac
	if *command-list* then *command-list else-part* fi
pipeline:	***command***
	pipeline* | *command
andor:	***pipeline***
	andor* && *pipeline
	andor* || *pipeline
command-list:	***andor***
	command-list ;
	command-list &
	command-list ; andor
	command-list & andor

input-output:	**>** ***file***
	< ***file***
	<< ***word***
	>> ***file***
	digit **>** ***file***
	digit **<** ***file***
	digit **>>** ***file***
file:	***word***
	& ***digit***
	& –
case-part:	***pattern*** **)** ***command-list*** **;;**
pattern:	***word***
	pattern **|** ***word***
else-part:	**elif** ***command-list*** **then** ***command-list else-part***
	else ***command-list***
	empty
empty:	
word:	a sequence of nonblank characters
name:	a sequence of letters, digits, or underscores starting with a letter
digit:	0 1 2 3 4 5 6 7 8 9

Metacharacters and Reserved Words

1. Syntactic

|	Pipe symbol
&&	And-if symbol
||	Or-if symbol
;	Command separator
;;	Case delimiter
&	Background commands
( )	Command grouping
<	Input redirection
<<	Input from a here document
>	Output creation
>>	Output append
#	Comment to end of line

2. Patterns

*	Match any character(s) including none
?	Match any single character
[...]	Match any of enclosed characters

3. Substitution

${...}	Substitute shell variable
\`...\`	Substitute command output

4. Quoting

\	Quote next character as literal with no special meaning
´...´	Quote enclosed characters excepting the back quotation marks (´)
"..."	Quote enclosed characters excepting: $ \` \ "

5. Reserved words

if	esac
then	for
else	while
elif	until
fi	do
case	done
in	{ }

Chapter 9
The C shell

The C shell program, **csh**(C) is a command language interpreter. The C shell, like the standard UNIX system shell **sh**(C,) is an interface between you and the UNIX operating system commands and programs. It translates command lines entered at a terminal into corresponding system actions, gives you access to information, such as your login name, home directory, and mailbox, and lets you construct shell procedures for automating system tasks.

This chapter explains how to use the C shell. It also explains the syntax and function of C shell commands and features, and shows how to use these features to create shell procedures. The C shell is fully described in **csh**(C) in the *User's Reference.*

Invoking the C shell

You can invoke the C shell from another shell by using the **csh** command. To invoke the C shell, enter the following command at the standard shell's command line:

```
csh
```

You can also direct the system to invoke the C shell for you when you log in. If the C shell is your login shell in your */etc/passwd* file entry, the system automatically starts the shell when you log in.

After the system starts the C shell, the shell searches your home directory for the command files *.cshrc* and *.login.* If the shell finds the files, it executes the commands contained in them, then displays the C shell prompt (%).

The *.cshrc* file typically contains the commands you want to execute each time you start a C shell, and the *.login* file contains the commands you want to execute after logging in to the system. For example, the following is the contents of a typical *.login* file:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
set time=15
set history=10
mail
```

This file contains several **set** commands. The **set** command is executed directly by the C shell; there is no corresponding UNIX system program for this command. This command sets the C shell variable **ignoreeof** which shields the C shell from logging out if you press ⟨Ctrl⟩d. Instead of ⟨Ctrl⟩d, use the **logout** command to log out of the system. Set the **mail** variable to tell the C shell to watch for incoming mail and notify you if new mail arrives.

Next, the C shell variable **time** is set to 15 causing the C shell to automatically print out statistics lines for commands that execute for at least 15 seconds of CPU time. The **history** variable is set to 10 indicating that the C shell remembers the last 10 commands typed in its history list (this is described later in this chapter).

Finally, the UNIX operating system **mail** program is invoked.

When you log out (by entering the **logout** command), the C shell executes commands from the file *.logout* if it exists in your home directory. After that, the C shell terminates and logs you off the system.

Using shell variables

The C shell maintains a set of variables. For example, in the above section, the variables **history** and **time** have the values 10 and 15. Each C shell variable has as its value an array of zero or more strings. You can assign values to C shell variables using the **set** command. The syntax for **set** is:

set ***name*** **=** ***value***

You can also use C shell variables to store values to use later in commands through a substitution mechanism. The most commonly referenced variables are those to which the C shell itself refers. By changing the values of these variables you can directly affect the behavior of the C shell.

One of the most important variables is **path**. This variable contains a list of directory names. When you enter a command name at your terminal, the C shell examines each named directory in turn, until it finds an executable file whose name corresponds to the name you entered.

The **set** command with no arguments displays the values of all variables currently defined in the C shell. The following example file shows typical default values:

```
argv    ()
home    /usr/bill
path    (. /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
```

This output indicates that the variable **path** begins with the current directory indicated by dot (.), then */bin*, and */usr/bin*. Your own local commands can be in the current directory. Normal UNIX system commands reside in */bin* and */usr/bin*.

Sometimes a number of locally developed programs reside in the directory */usr/local*. If you want all C shells that you invoke to have access to these new programs, place the following command in the *.cshrc* file in your home directory:

```
set path=(. /bin /usr/bin /usr/local)
```

Try doing this, then logging out and back in. To see that the value assigned to **path** changed, enter:

```
set
```

You should be aware that when you log in the C shell examines each directory that you insert into your path and determines which commands are contained there. However, the C shell treats the current directory specially. This means that if you add commands to a directory in your search path after you have started the C shell, they might not necessarily be found. If you want to use a command which has been added after you have logged in, enter the following command:

```
rehash
```

The **rehash** command causes the shell to recompute its internal table of command locations, so that it finds the newly added command. Since the C shell has to look in the current directory on each command anyway, placing it at the end of the path specification usually works best and reduces overhead.

Other useful built-in variables are **home** which shows your home directory, and **ignoreeof** which (when set in your *.login* file) tells the C shell not to exit when it receives an end-of-file from a terminal. The variable **ignoreeof** is one of several variables whose value the C shell does not care about; the C shell is only concerned with whether these variables are set or unset. Thus, to set **ignoreeof**, simply enter:

```
set ignoreeof
```

To unset it, enter:

```
unset ignoreeof
```

Some other useful built-in C shell variables are **noclobber** and **mail**.

Set the **noclobber** variable to prevent accidental overwriting of files. Normally, if you redirect the standard output of a command to a file like this:

command > filename

you overwrite and destroy the previous contents of the named file. In this case, you might accidentally overwrite a valuable file. If you prefer that the C shell not overwrite files in this way, set **noclobber** in your *.login* file:

```
set noclobber
```

Now, when you enter a command like the following:

```
date > now
```

if the file *now* already exists, the shell displays an error message. If you really want to overwrite the contents of *now*, you can enter:

```
date >! now
```

The >! is a special syntax indicating that overwriting or "clobbering" the file is OK. (The space between the exclamation point (!) and the word "now" is critical here, as "!now" invocates the history mechanism, and has a completely different effect.)

Using the C shell history list

The C shell can maintain a history list of previously entered commands. With this list, you can use a notation that reuses commands, or words from commands, to form new commands. You can use this mechanism to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the C shell (boldface indicates user input):

```
% cat bug.c
main()
{
    printf("hello);
}
% cc !$
cc bug.c
bug.c(4) :error 1: newline in constant
% ed !$
ed bug.c
28
3s/);/"&/p
    printf("hello");
w
29
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
29
3s/lo/lo\\n/p
    printf("hello\n");
w
31
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 5124 + 614 + 1254 = 6692 = 0x1b50
bug: 5124 + 616 + 1252 = 6692 = 0x1b50
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill   7648 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill   7650 Dec 19 09:42 bug
% bug
hello
% pr bug.c | lpt
lpt: Command not found.
% ^lpt^lpr
pr bug.c | lpr
%
```

In this example, we have a very simple C program that has a bug or two in the file *bug.c*; we use **cat**(C) to display it on the terminal. We then try to run the C compiler on it, referring to the file again as **!$** meaning the last argument to the previous command. Here the exclamation mark (**!**) is the history mechanism invocation metacharacter, and the dollar sign (**$**) stands for the last argument, by analogy to the dollar sign in the editor which stands for the end-of-line.

The C shell echoes the command (as you would have typed it and then executes the command. The compilation yielded error diagnostics, so we edit the file, fix the bug, and run the C compiler again, this time referring to this command simply as **!c**, which repeats the last command that started with the letter "c."

If there are other commands that begin with the letter "c" executed recently, you can use **!cc** or even **!cc:p**. The **:p** allows you to print the last command starting with "cc" without executing it, so that you can check to see whether you really want to execute a given command.

After recompiling, we ran the resulting *a.out* file, and then noting that there was still a bug, ran the editor again. After fixing the program we ran the C compiler again, but included the **-o bug** option, telling the compiler to place the resultant binary in the file *bug* rather than *a.out*. (In general, you can use the history mechanisms anywhere to form new commands, and you can place other characters before and after the substituted commands.)

We then ran the **size** command to see how large the binary program images are, and then we ran an **ls -l** command with the same argument list, denoting the argument list:

```
!*
```

Finally, we ran the program *bug* to see that its output is indeed correct.

To make a listing of the program, we ran the **pr** command on the file *bug.c*. In order to print the listing at a lineprinter, we piped the output to **lpr**, but misspelled it as "lpt." To correct and rerun the command, we used a C shell substitute, placing the old text and new text between caret (^) characters. This is similar to the substitute command in the editor.

There are other mechanisms available for repeating commands. The **history** command prints out a numbered list of previous commands. You can then refer to these commands by number. You can also refer to a previous command by searching for a string which appeared in it. See the **csh**(C) manual page in the *User's Reference* for a complete description of all these mechanisms.

Using aliases

The C shell has an alias mechanism that you can use to change the string that you use to enter the command. This mechanism can be used to simplify the commands you enter, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features of aliasing can also be obtained by using C shell command files. However, these take place in another instance of the C shell and cannot directly affect the current C shell's environment or involve commands such as **cd** which must be done in the current C shell.

For example, suppose there is a new version of the mail program on the system called *newmail* that you wish to use instead of the standard mail program *mail*. If you place the C shell command

```
alias mail newmail
```

in your *.cshrc* file, when you enter the following command:

```
mail bill
```

the C shell runs the **newmail** program. Suppose you want the command **ls** to display sizes of files. In other words, you want **ls** to use the **-s** option always. In this case, you can use the **alias** command like this:

```
alias ls ls -s
```

If you are used to DOS, you might create the following alias to perform the same thing:

```
alias dir ls -s
```

Now, if you enter:

```
dir ~bill
```

C shell translates this to:

```
ls -s /usr/bill
```

Note that the tilde character (~) is a special C shell symbol that represents the user's home directory.

Thus, you can use **alias** to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. You can also define aliases that contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism.

For example, the following definition:

```
alias cd 'cd \!* ; ls '
```

specifies to run an **ls** command after each **cd** command. We enclosed the entire alias definition in single quotation marks (') to prevent most substitutions from occurring and to prevent the semicolon character (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from being interpreted when the alias command is entered. The \!* here substitutes the entire argument list to the pre-aliasing **cd** command; no error is given if there are no arguments. The semicolon separating commands is used here to indicate that one command is to be done and then the next. Similarly, the following example defines a command that looks up its first argument in the password file:

```
alias whois 'grep \!^ /etc/passwd'
```

The C shell currently reads the *.cshrc* file each time it starts up. If you place a large number of aliases there, C shells tend to start slowly. You should try to limit the number of aliases you have to a reasonable number (10 or 15 is reasonable). Too many aliases causes delays and makes the system seem sluggish when you execute commands from within an editor or other programs.

Redirecting input and output

In addition to the standard output, commands also have a diagnostic output that is normally directed to the terminal even when the standard output is redirected to a file or a pipe. Sometimes, you might want to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long running command into a file and want to have a record of any error diagnostic it produces, you can enter:

```
command >& file
```

The >& here tells the C shell to route both the diagnostic output and the standard output into *file*. Similarly, you can enter the following command to route both standard and diagnostic output through the pipe to the lineprinter:

```
command |& lpr
```

Use the following form when **noclobber** is set and *file* already exists:

```
command >&! file
```

Finally, use the following form to append output to the end of an existing file:

```
command >> file
```

If **noclobber** is set, an error results if *file* does not exist, otherwise the C shell creates *file*. The following form allows you to append to a file even if it does not exist and **noclobber** is set:

```
command >>! file
```

Creating background and foreground jobs

Usually, every line entered to the C shell creates a job. Single commands without pipes or semicolons create the simplest jobs. When you enter one or more commands together as a pipeline or as a sequence of commands separated by semicolons, the C shell creates a single job consisting of these commands together as a unit. Each of the following lines creates a job:

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If you enter the ampersand metacharacter (&) at the end of the commands, the job is started as a background job. This means that the C shell does not wait for the job to finish, but instead, immediately prompts for another command. The job runs in the background at the same time that normal jobs, called foreground jobs, continue to be read and executed by the C shell. Thus, the following command:

```
du > usage &
```

runs the **du** program (which reports on the disk usage of your working directory), puts the output into the file *usage*, and returns immediately with a prompt for the next command without waiting for *du* to finish. The *du* program continues executing in the background until it finishes; you can enter and execute more commands in the mean time. Background jobs are unaffected by any signals from the keyboard such as the INTERRUPT or QUIT signals.

The **kill** command terminates a background job immediately. Normally, this is done by specifying the process number of the job to be killed. You can determine the process number of a job with the **ps** command. See Chapter 5, "Managing processes," for more information.

Using built-in commands

This section explains how to use some of the built-in commands.

Use the **alias** command described above to assign new aliases and to display existing aliases. If given no arguments, **alias** prints the list of current aliases. You can also give **alias** one argument in order to show the current alias for a given string of characters. For example, the following command prints the current alias for the string **ls**:

```
alias ls
```

The **history** command displays the contents of the history list. You can use the numbers given with the history events to reference previous events that are difficult to reference contextually. There is also a C shell variable named **prompt**. By placing an exclamation point (!) in its value, the C shell substitutes the number of the current command in the history list. You can use this number to refer to a command in a history substitution. For example:

```
set prompt='\! % '
```

Note that the exclamation mark has to be escaped here even within back quotes.

The **logout** command is used to terminate a login C shell that has **ignoreeof** set.

The **rehash** command causes the C shell to recompute a table of command locations. This is necessary if you add a command to a directory in the current C shell's search path and want the C shell to find it, because otherwise the hashing algorithm might tell the C shell that the command was not in that directory when the hash table was computed.

Use the **repeat** command to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five*, you enter:

```
repeat 5 cat one >> five
```

Use the **setenv** command to set variables in the environment. Thus, to set the value of the environment variable **TERM** to **adm3a**, use this command:

```
setenv TERM adm3a
```

Use the **env** program to print out the environment. For example, the output from **env** might look like this:

```
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
```

Use the **source** command to force the current C shell to read commands from a file. Thus, use the following command after editing the *.cshrc* file to read in the changes that you want to take effect before the next time you login:

```
source .cshrc
```

The same holds true when using the **source** command with the *.login* file.

Use the **time** command to time a command regardless of how much CPU time it takes. Thus, the following command:

```
time cp /etc/termcap /usr/bill/termcap
```

displays:

```
0.0u 0.4s 0:02 21%
```

Similarly, this command:

```
time wc /etc/termcap /usr/bill/termcap
```

displays:

```
   2071   5849  92890 /etc/termcap
   2071   5849  92890 /usr/bill/termcap
   4142  11698 185780 total
1.3u 0.7s 0:04 47%
```

This output indicates that the **cp** command used a negligible amount of user time (u) and about 4/10ths of a second of system time (s); the elapsed time was 2 seconds (0:02). The word count command **wc** used 1.3 seconds of user time and 0.7 seconds of system time in 4 seconds of elapsed time. The percentage "47%" indicates that over the period when it was active, the **wc** command used an average of 47 percent of the available CPU cycles of the machine.

Use the **unalias** and **unset** commands to remove aliases and variable definitions from the C shell.

Creating shell scripts

You can place commands in files and tell the C shell to read and execute commands from these files, called "C shell scripts" or simply "shell scripts." This section describes the C shell features that are useful when creating shell scripts.

Using the argv variable

To interpret a **csh** command script, use a command like the following:

csh ***script argument*...**

where ***script*** is the name of the file containing the C shell commands and ***argument*** is a sequence of command arguments. The C shell places these arguments in the **argv** variable and then begins to read commands from ***script***. These parameters are then available through the same mechanisms that you use to reference any other C shell variables.

To run your shell script in the file called *script*, first place a C shell comment at the beginning of the shell script (i.e., begin the file with a number sign (#)). Then, make *script* executable by entering:

```
chmod 755 script
```

or:

```
chmod +x script
```

Now, when you enter the following command, */bin/csh* is invoked automatically to execute *script*:

```
script
```

If the file does not begin with a number sign (#), the standard shell */bin/sh* is used to execute it. (For more information about **sh**, see Chapter 8, "The Bourne shell."

Substituting shell variables

After each input line is broken into words and history substitutions are performed on it, the input line is parsed into distinct commands. Before each command is executed, a mechanism known as variable substitution is performed on these words. Keyed by the dollar sign ($), this substitution replaces the names of variables with their values. Thus, when you place the following command in a shell script:

echo $argv

the current value of the **argv** variable is echoed to the output of the shell script. If **argv** is unset at this point, the C shell displays an error.

A number of notations are provided for accessing components and attributes of variables. The notation:

$?*name*

expands to 1 if ***name*** is set or to 0 if ***name*** is not set. This is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The following notation expands to the number of elements in the variable ***name***:

$#*name*

To illustrate this, examine the following terminal session (user input appears in boldface):

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

You can also access the components of a variable that has several values. Thus:

$argv[1]

gives the first component of **argv** (in the example above, "a"). Similarly:

$argv[$#argv]

gives "c". Other notations useful in C shell scripts are:

$*n*

where ***n*** is an integer. This is shorthand for:

$argv[*n* **]**

(the ***n***'th parameter). The following notation:

$*

is shorthand for:

$argv

The form:

$$

expands to the process number of the current C shell. Because this process number is unique in the system, it is often used to generate unique temporary filenames.

One minor difference between **$*n*** and **$argv[*n*]** should be noted here. The form: **$argv[*n*]** yields an error if ***n*** is not in the range 1-**$#argv** while **$n** never yields an out-of-range subscript error. This is for compatibility with the way older shells handle parameters.

Another important point is that, if there are less than ***n*** components of the given variable and you give a subrange of the form:

***n*-**

it is never an error and no words are substituted. Likewise, a range of the form:

m*-*n

returns an empty vector without giving an error when ***m*** exceeds the number of elements of the given variable, provided the subscript ***n*** is in range.

Using expressions

To construct useful C shell scripts, the C shell must be able to evaluate expressions based on the values of variables. In fact, all the arithmetic operations of the C language are available in the C shell with the same precedence that they have in C. In particular, the operations == and != compare strings and the operators **&&** and **| |** implement the logical **AND** and **OR** operations.

The C shell also allows file inquiries of the following form:

> **-?** ***filename***

where question mark (?) is replaced by a number of single characters. For example, the expression primitive:

> **-e** ***filename***

tells whether ***filename*** exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or if it has nonzero length.

It is possible to test whether a command terminates normally, by using a primitive of the form:

> { ***command*** }

This returns 1 if the command exits normally with exit status 0, or 0 if the command terminates abnormally or with a nonzero exit status. If more detailed information about the execution status of a command is required, it can be executed and you can examine the **status** variable in the next command. Because **$status** is set by every command, its value is always changing.

For the full list of expression components, see **csh**(C) in the *User's Reference.*

A sample script

A sample shell script follows that uses the expression mechanism of the C shell and some of its control structures:

```
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i != *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script uses the **foreach** command, which iteratively executes the group of commands between the **foreach** and the matching **end** statements for each value of the variable **i**. If you want to look more closely at what happens during execution of a **foreach** loop, you can use the debug command **break** to stop execution at any point. To resume execution, use the debug command **continue**. The value of the iteration variable (*i* in this case) remains at whatever it was when the last **foreach** loop was completed.

The **noglob** variable is set to prevent filename expansion of the members of **argv**. In general, this is a good idea if the arguments to a C shell script are filenames which have already been expanded or if the arguments might contain filename expansion metacharacters. You can also quote each use of a **$** variable expansion, but this is harder and less reliable.

The other control construct is a statement of the form:

> **if (** ***expression*** **) then**
>
> ***command***
>
> . . .
>
> **endif**

The placement of the keywords in this statement is not flexible due to the current implementation of the C shell. For example, the following two formats are not acceptable to the C shell:

```
if ( expression ) # Won't work!
then
        command
        . . .
endif
```

and:

```
if (expression) then command endif # Won't work
```

The C shell does have another form of the **if** statement:

```
if ( expression ) command
```

You can also write this **if** statement this way:

```
if ( expression ) \
        command
```

In this example, we escape the newline for the sake of appearance. The command must not include | , &, or ; and must not be another control command. The second form requires the final backslash (\) immediately preceding the end-of-line.

The more general **if** statements above also admit a sequence of **else-if** pairs followed by a single **else** and an **endif**. For example:

```
if ( expression ) then
        commands
else if ( expression ) then
        commands
. . .
else
        commands
endif
```

Another important mechanism used in C shell scripts is the colon (:) modifier. For example, use the **:r** modifier to extract the root of a filename or **:e** to extract the extension. Thus if the variable **i** has the value */mnt/foo.bar,* then, the following command:

echo $i $i:r $i:e

produces:

```
/mnt/foo.bar /mnt/foo bar
```

This example shows how the **:r** modifier strips off the trailing ".bar" and the **:e** modifier leaves only the "bar." Other modifiers take off the last component of a pathname leaving the head **:h** or all but the last component of a pathname leaving the tail **:t**. These modifiers are fully described in the **csh**(C) page in the *User's Reference*. You can also use the command substitution mechanism to perform modifications on strings to then re-enter the C shell environment. Because each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. It is also important to note that the current implementation of the C shell limits the number of colon modifiers on a **$** substitution to 1. Thus, the following command:

echo $i $i:h:t

produces:

```
/a/b/c /a/b:t
```

and does not do what you might expect.

Finally, note that the number sign character (#) lexically introduces a C shell comment in C shell scripts (but not from the terminal). The C shell discards all subsequent characters on the input line after a number sign. You can quote this character using ´ or \ to place it in an argument word.

Using other control structures

The C shell also has control structures **while** and **switch** similar to those of C. These take the forms:

```
while ( expression )
        commands
end
```

and:

```
switch ( word )

case str1:
        commands
        breaksw

...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the **csh**(C) manual page in the *User's Reference.* C programmers should note that **breaksw** exits from a **switch** and **break** exits a **while** or **foreach** loop. A common mistake to make in C shell scripts is to use **break** rather than **breaksw** in switches.

Finally, the C shell includes a **goto** statement, with labels looking like they do in C:

```
loop:
        commands
        goto loop
```

Supplying input to commands

Commands run from C shell scripts receive by default the standard input of the C shell which is running the script. This allows C shell scripts to fully participate in pipelines, but mandates extra notation for commands that are to take inline data.

Thus, you need a metanotation for supplying inline data to commands in C shell scripts. For example, consider this script which runs the editor to delete leading blanks from the lines in each argument file:

```
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << ' EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
```

The notation:

```
<< 'EOF'
```

means that the standard input for the **ed** command is to come from the text in the C shell script file up to the next line consisting of exactly **EOF**. The fact that the **EOF** is enclosed in single quotation marks (´), i.e., it is quoted, causes the C shell not to perform variable substitution on the intervening lines. In general, if any part of the word following the << (which the C shell uses to terminate the text to be given to the command) is quoted, these substitutions are not performed. In this case, because we use the form **1,$** in our editor script, we need to insure that this dollar sign is not variable substituted. We can also insure this by preceding the dollar sign ($) with a backslash (\ ), i.e.:

1,\$s/^[ ]*//

Quoting the **EOF** terminator is a more reliable way of achieving the same thing.

Catching interrupts

If a C shell script creates temporary files, you might want to catch interruptions of the C shell script so that you can clean up these files. To do this:

onintr ***label***

is included in the script, where ***label*** is a label in your program. If an interrupt is received, the C shell performs a **goto label** and you can remove the temporary files and run the **exit** command (which is built into the C shell) to exit the C shell script. If you want to exit with nonzero status, use the following command to exit with status 1:

exit (1)

Using other features

There are other features of the C shell that shell script writers might find useful. Use the **verbose** and **echo** options and the related **-v** and **-x** command-line options to help trace the actions of the C shell. The **-n** option causes the C shell only to read commands and not to execute them.

One other thing to note is that the C shell does not execute C shell scripts that do not begin with the number sign character (#).

The C shell also includes another quotation mechanism using the double quotation mark ("), that allows only some of the expansion mechanisms to occur on the quoted string. It serves to make this string into a single word as the single quote (´) does.

Starting a loop at a terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if there are three shells in use on a particular system, */bin/sh, /bin/nsh,* and */bin/csh,* you can count the number of persons using each shell with the following commands:

```
grep -c csh$ /etc/passwd
grep -c nsh$ /etc/passwd
grep -c -v /sh$ /etc/passwd
```

These commands are very similar, but you can use **foreach** to simplify them (boldface indicates user input):

```
% foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
```

Note here that the C shell prompts for input with " ? " when reading the body of the loop. This occurs only when the **foreach** command is entered interactively.

Variables that contain lists of filenames or other words are also useful with loops. For example, examine the following terminal session:

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
```

The **set** command here gives the variable **a** a list of all the filenames in the current directory as value. You can then iterate over these names to perform any chosen function.

The C shell converts the output of a command within back quotation marks (`) to a list of words. You can also place the quoted string within double quotation marks (") to take each (nonempty) line as a component of the variable. This prevents the lines from being split into words at blanks and tabs. Use the :x modifier to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

Using braces with arguments

Another form of filename expansion involves the characters, { and }. These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus:

A{str1,str2,...strn}B

expands to:

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and can be applied recursively (i.e., nested). The results of each expanded string are sorted separately, preserving left to right order. If no other expansion mechanisms are used, the resulting filenames do not have to exist. This means that you can use this mechanism to generate arguments which are not filenames, but which have common parts.

For example, use this to make subdirectories *hdrs, retrofit,* and *csh* in your home directory:

mkdir ~/{hdrs,retrofit,csh}

This mechanism is most useful when the common prefix is longer; for example:

chown root /usr/demo/{file1,file2,...}

Substituting commands

A command enclosed in accent symbols (`) is replaced, just before filenames are expanded, by the output from that command. Thus, use this command to save the current directory in the variable **pwd**:

```
set pwd=`pwd`
```

Another example runs the **vi** editor, supplying as arguments those files whose names end in *.c* that contain the string "TRACE":

```
vi `grep -l TRACE *.c`
```

Command expansion also occurs in input redirected with << and within quotation marks ("). Refer to **csh**(C) in the *User's Reference* for more information.

Special characters

The following table lists the **csh** and UNIX operating system special characters. A number of these characters also have special meaning in expressions. See the **csh** manual section for a complete list.

;	separates commands to be executed sequentially
|	separates commands in a pipeline
( )	brackets expressions and variable values
&	follows commands to be executed without waiting for completion

Filename metacharacters

/	separates components of a file's pathname
.	separates root parts of a filename from extensions
?	expansion character matching any single character
*	expansion character matching any sequence of characters
[ ]	expansion sequence matching any single character from a set of characters
~	used at the beginning of a filename to indicate home directories
{ }	used to specify groups of arguments with common parts

Quotation metacharacters

\	prevents meta-meaning of following single character
´	prevents meta-meaning of a group of characters
"	like ´, but allows variable and command expansion

Input/output metacharacters

<	indicates redirected input
>	indicates redirected output

Expansion/Substitution metacharacters

$	indicates variable substitution
!	indicates history substitution
:	precedes substitution modifiers
^	used in special forms of history substitution
`	indicates command substitution

Other metacharacters

#	begins scratch filenames; indicates C shell comments
-	prefixes option (flag) arguments to commands

Chapter 10

The Korn shell

This chapter contains an introduction to the Korn shell, **ksh**(C), a powerful tool that acts as a control center for you in your day to day work with the operating system.

The **ksh** is provided as an alternative to the **sh** and **csh** programs. It contains almost all the features of **sh**, along with many new facilities. Consequently, if you are new to the operating system you should read the chapter titled "The Bourne shell" before starting this section.

Overview

The Korn shell is an interactive command language interpreter and programming language, which reads and executes commands from either the terminal or a file.

It is command compatible with the Bourne shell, and has history and substitution features similar to those of the C shell. It also provides command line editing, job control and enhanced command history functions.

The Korn shell maintains a history file of previously executed command lines. Commands can be retrieved and edited using the same keystrokes as the **vi** or **emacs** editors. You can search for and substitute previously executed commands and parameters using the text search and replacement facilities of the selected editor.

Extensive job control features are built into the Korn shell. Using these facilities you can start tasks running in the background, adjust the priority of executing tasks, move background tasks into the foreground, cause tasks to wait for a fixed period, and kill tasks. These features are also available to shell scripts executing in the Korn shell environment.

The Korn shell command language is an extended version of the language recognized by the Bourne shell (**sh**(C)). Extensive command aliasing and substitution features are provided, similar to those of the C shell. In addition, the Korn shell provides a rich set of expressions and program control structures. All the job control and command substitution features are available to a shell script executing under the shell. Consequently, the Korn shell language provides greater flexibility and power than either the Bourne shell or the C shell, while maintaining compatibility with the Bourne shell.

Setting up the Korn shell

If the Korn shell is not installed as the default login shell, it can be started from the command line of either of the other shells, like any other program, by typing its name:

```
ksh ⟨Enter⟩
```

If no arguments are specified, **ksh** loads and runs in interactive mode; it displays a prompt, and any commands typed at the console are executed immediately. If **ksh** is started with the name of a file as an argument, and the file is marked as being executable, **ksh** checks the type of the file and executes it as a script (if it contains text) or runs it as a program (if it is a binary file).

For serious use, it is normal to install **ksh** as the default login shell. To change your login shell, ask your system administrator to change your login shell specifier in the */etc/passwd* file to **ksh**.

When the system administrator specifies that **ksh** is to be used as the login shell, the **sysadmsh**(ADM) utility creates two files in the user's home directory; *.profile* and *.kshrc*.

When you log in, **ksh** reads commands from the system profile file, */etc/profile*, then from *.profile* in the current directory (or **$HOME**/*.profile*, if either file exists). The *.profile* file is used to set and export variables that are to be applied to the entire work session.

For example, it is possible to use *.profile* to set and export the **TERM** variable, which specifies the type of terminal you are using in the current session. (This is recognized by other applications and shells.)

You can use *.profile* to set a trap that causes **ksh** to execute another file, for example, *.logout*, when you exit the shell. To do this, enter the following in *.profile*:

```
trap $HOME/.logout 0
```

The *.logout* file must be executable.

NOTE This particular example is only useful to provide backward compatibility with **csh**. Otherwise, there is no reason to use this trap.

After executing *.profile,* the shell reads commands from the **ksh** environment file, **$HOME/***.kshrc* (if it exists). *.kshrc* is used to customize the working environment for **ksh** sessions; it is not executed by the other shells. Commands and definitions that only **ksh** recognizes should therefore be defined in this file rather than in *.profile,* which is used by **sh** and **csh**.

For example, the **ignoreeof** variable prevents **ksh** from exiting when it reads an end of file character; this stops the user from logging out by hitting ⟨Ctrl⟩**d**. Because **ignoreeof** is only recognized by **ksh**, it should be set in *.kshrc* rather than in *.profile.*

After reading *.kshrc,* **ksh** looks for the *.sh_history* file. This file contains a list of previously entered commands; if there is no *.sh_history* file in the home directory, **ksh** creates one. (For more information on the *.sh_history* file, refer to the section below on additional editing features.)

Flags and options for ksh

A large number of startup flags may be specified when ksh is started from the command line. Additionally, environment variables can be set to configure the **ksh** operating parameters. For a full list of all the flags and parameters which the shell recognizes, see **ksh**(C).

It is not normally necessary to change the startup flags associated with **ksh**: however you may want to set some of the **ksh** environment variables in your *.profile* file. To assign values to **ksh** environment variables, use this format:

```
EDITOR=/bin/vi
export EDITOR
```

The first line assigns the value **/bin/vi** to the environment variable **EDITOR** (which is created by the assignment expression if it does not already exist). The second line contains the **export** command, which specifies that the **EDITOR** variable is to be exported in the environment that **ksh** passes to any tasks that it starts. (If you do not export variables, they are unavailable to secondary shells or programs invoked from **ksh**. Because variables often convey essential information about the environment in which programs run, this is generally undesirable.)

The following table shows some of the more common environment variables:

Table 10-1 Environment variables

Variable	Description
COLUMNS	specifies the number of columns that **ksh** uses to display the command line
EDITOR	sets either the **vi** like or **emacs** like editor to use when editing the command line. (EDITOR must not be set if **ksh** is used on the console.)
ENV	sets the environment file (*.kshrc* by default)
HISTFILE	sets an alternate history file (*.sh_history* by default)
HISTSIZE	sets the maximum number of commands that are stored in the history file (128 by default)
HOME	specifies the default argument used by the **cd**(C) command
MAILCHECK	specifies the interval in seconds that **ksh** checks for new mail, if MAIL is set in *.profile*. The default value is 600 seconds.
PATH	specifies the pathnames that **ksh** searches when executing commands
PS1	specifies the primary prompt to display when the **interactive** option is on ($ by default); **ksh** replaces an exclamation point (!) with the command number (to print a ! in the prompt, enter !!)
TERM	specifies the type of terminal that you are using
VISUAL	sets the editor to use when editing command lines (overrides the value of EDITOR). VISUAL must not be set if ksh is used on the console.

Editing features

Using **csh** or **sh**, the only way to fix errors on the command line is to backspace or retype the entire line. With **ksh** you can edit the command line using the same commands that you use to edit files. You can move through the commands in your history file (the list of commands you have already typed at the console) and re-execute commands. You can also edit command lines, copying sections of other commands into your current command line. The Korn shell provides both vi-like and emacs-like built in editor interfaces for manipulating the command line.

The *.kshrc* startup file normally contains a command which turns on the vi-like editor. In this mode, **ksh** responds to the editing commands used by **vi**. You can turn off the **ksh** editor completely, or turn off **vi** and turn on **emacs** emulation for the current session or for every login session.

To turn off **vi** for the current login session only, enter the following at the command line:

set +o vi

To turn on **emacs** for the current login session, enter the following at the command line:

set -o emacs

(It is necessary to turn one editor off before the other can be turned on.) To turn either editor on or off automatically when you log in, add the appropriate command

set [+/-] [editorname]

to the *.profile* or environment file (*.kshrc* by default).

You can use the **EDITOR** and **VISUAL** environment variables to set the editor to any pathname that ends in **vi** (or **emacs**). For example, to turn on the **vi** editor automatically when you log in, add the following line to your *.kshrc* file:

EDITOR=/usr/bin/vi

Note that the **VISUAL** variable overrides the **EDITOR** variable.

Using the vi built-in editor mode

Like the **vi** text editor, **ksh**'s built in **vi** editor has two modes; input mode and control mode. In input mode, **ksh** inserts the characters you type at the keyboard into an editing buffer. In control mode, **ksh** interprets the characters you type at the keyboard as editing commands.

When you log in using **ksh** as your login shell, you are in input mode by default, with the cursor positioned at the beginning of the line. This differs from **vi**, where you are initially in control mode by default, and must press **a** or **i** to begin entering text. To enter control mode from input mode, press ⟨Esc⟩. If you press ⟨Esc⟩ when you are already in control mode, the terminal beeps.

Editing in vi input mode

While entering commands in input mode, you can edit the current command line using editing commands from the following table:

Table 10-2 Input mode editing commands

Command	Description
⟨Ctrl⟩**h** or ⟨Bksp⟩	moves back one character
⟨Return⟩ or ⟨Ctrl⟩**m**	executes the current line
⟨Ctrl⟩**v**	escapes the character that follows (for entering control characters)

Editing in control mode

At any time before you press ⟨Return⟩ to execute the command, you can press ⟨Esc⟩ to enter control mode. In control mode, you can move around the command line as if you were in **vi**, editing a file.

The following table shows the **vi** commands for moving the cursor on the command line in control mode:

Table 10-3 Moving the cursor

Key	Description
h	moves left one character
l	moves right one character
b	moves left one word
B	moves left one word, skipping punctuation
w	moves right one word
W	moves right one word, skipping punctuation
e	moves to the last character of the next word
E	moves to the last character of the next word, skipping punctuation
0	moves to the beginning of the current line
$	moves to the end of the current line
^	moves to the first character on the current line that is not a ⟨Space⟩ or ⟨Tab⟩
f*x*	moves right to the next occurrence of *x*
F*x*	moves left to the preceding occurrence of *x*
t*x*	moves right to the character before the next occurrence of *x*
T*x*	moves left to the character following the preceding occurrence of *x*
;	repeats the last character search **f**, **F**, **t**, or **T**.
,	reverses the last character search **f**, **F**, **t**, or **T**.

The following table gives the commands for entering input mode from control mode, and for changing and deleting text:

Table 10-4 Adding changing and deleting text

Key	Description
a	enters input mode after the character under the cursor
A	enters input mode after the last character on the line
i	enters input mode before the character under the cursor
I	enters input mode before the first character on the line
_	appends the last word of the previous **ksh** command to the current line and then enters input mode.
rz	replaces the character under the cursor with *z*
R*text*	replaces characters with *text* beginning at the cursor
c*motion*	changes the characters from cursor position, using the **vi** *motion* command For example:
cw	changes word below cursor
cl	changes character below cursor and then adds text
c$	changes from the current character to the end-of-line
cc	deletes the entire line and returns to input mode (same as c$)
x	deletes the character under the cursor
X	deletes the character to the left of the cursor
dw	deletes the word under the cursor
d*motion*	deletes characters, starting at the cursor, up to and including the other end of *motion*
D	deletes from the cursor to the end of line
d$	same as **D**
dd	deletes the entire line
y*motion*	yanks the current character using the **vi** motion command
Y	yanks from cursor to end of line
y$	same as **Y**
yy	yanks the entire line into the buffer
p	puts previously yanked (or deleted) words to the right of the cursor
P	puts previously yanked (or deleted) words to the left of the cursor

The following table shows the control mode commands for executing and redrawing the current line, repeating commands, and undoing modifications on the command line:

Table 10-5 Miscellaneous control mode commands

Command	Description
⟨Return⟩ or ⟨Ctrl⟩ **m**	executes the current line
⟨Ctrl⟩ **l**	redraws the current line
~	changes the case of the character under the cursor
.	repeats the most recent **vi** command
u	undoes the previous **vi** command
U	undoes all modifications on the current line

Accessing commands in the history file

Using the **vi** built in editor, you can access previously entered commands that are stored in your **ksh** history file (*.sh_history*). Once you retrieve a command, you can modify it and execute it again.

This serves two main purposes. Firstly, it significantly reduces retyping when you are working with complex, long command lines. Secondly, because the commands are saved in *.sh_history*, you can copy the file and edit the copy to produce a shell script that repeats your commands, or embed them in a more complex program. (See the section "Programming the Korn shell.")

Displaying commands in the history file

To display the list of the commands that are stored in the history file, enter **history**. The **history** command is a predefined alias that uses the **ksh** built in command, **fc** (fix command), to access the history file. For more information about **fc**, see **ksh**(C).

The **history** alias displays the last 16 commands in the history file (or fewer, if there are fewer than 16 commands in the file). You can specify how many and which commands that you want **history** to display. Note that the commands must be accessible in the history file for **history** to display them. The following list gives examples of how to use the **history** alias:

history -4	displays the previous four commands only.
history 20	displays all commands from the history file, starting with 20.
history 12 24	displays only commands 12 through 24.
history >*fred*	copies all commands from the history file to a file called *fred*.

Re-executing previous commands

The **ksh** also includes **r**, another predefined alias that uses the **fc** built-in command. The **r** alias allows you to re-execute commands from the history file. This alias functions similarly to the **!** command in **csh**. The following list shows some common uses of the **r** alias:

r	re-executes the last command entered
r *command*	re-executes the last *command* entered
r *x*	re-executes the last command beginning with *x*
r #	re-executes command number #

Note that **r** simply re-executes commands from the history list; **r** does not allow you to modify commands before you execute them.

Editing previous commands

You can use **vi** commands to search for and retrieve commands from the history file. Once you locate a command, you can edit and re-execute it using the **vi** commands described in the section "Editing in control mode" earlier in this chapter.

To move through the history file, first press ⟨Esc⟩ to enter control mode. Then, use the **vi** commands in the following table to move up and down in the history file:

Table 10-6 Moving in the history file

Command	Description
k	moves up (previous) one command in the history file
j	moves down (next) one command in the history file
/*string*	searches left and up (back) through the history file for the next command containing *string*
?*string*	searches right and down (forward) through the history file for the next command containing *string*
G	goes back to the oldest accessible command in the history file
n	repeats the last / or ? search command
N	repeats the last / or ? command, searching backward

Configuring the history file

In addition to the command editing facilities provided by **ksh**, a number of other features are provided to make life easier. If you use a particular command frequently, you can define aliases for them. When the alias is typed, the original command is carried out instead; short, simple alias names can be used to invoke long, complex commands. (This feature is described in the section "Alias expansion.") You can also save your history file and use it in subsequent sessions in conjunction with your aliases. This enables you to use the command recall facilities of **ksh** to run complex or frequently used command lines in any session.

The **ksh** normally saves commands in the *sh_history* file in your home directory. This is a default setting: to specify a different history file, set the **HISTFILE** environment variable in your *.profile* file.

For example, to use the file *.history* instead of *.sh_history*, add the following lines to your *.profile*:

```
HISTFILE=~/.history
export HISTFILE
```

You can also specify the maximum number of previously entered commands that you can retrieve from the history file with the **HISTSIZE** environment variable. If **HISTSIZE** is not set, **ksh** stores 128 commands by default. There is no limit to the number of commands that **ksh** can store. If **HISTSIZE** is very large, **ksh** may be very slow at startup time.

The **ksh** does not delete the history file when you exit; the shell appends and stores commands across login sessions. When you log in, **ksh** deletes any commands in your history file that are older than the last number of commands specified by **HISTSIZE**.

Manipulating commands wider than the screen

The **ksh** allows you to enter commands of up to 256 characters from the terminal. You can define the maximum width of the command line display (80 columns by default) using the **COLUMNS** variable.

If you edit a command that is wider than the command line, **ksh** automatically scrolls the command line horizontally to the left or right of your screen. In the last column on the right side of the screen, **ksh** displays one of the following characters to show that the line is scrolling:

<	scrolls to the right (text to the left is not displayed)
>	scrolls to the left (text to the right is not displayed)
*	text both to the right and to the left is not displayed

For example, if **COLUMNS** is not set, the width of the command line display is the default of 80 columns and your command line is greater than 78 characters wide, **ksh** scrolls the command line left to display the end of the line. To the right of the command line,**ksh** displays the > character. It is possible to edit lines wider than the screen by using the **0** and **$** commands (start of line and end of line respectively) to jump to the appropriate point.

Using expanded cd capabilities

The **ksh** includes expanded functionality for the **cd**(C) command. You can instruct **ksh** to search through a specified list of directories when you enter pathnames that do not begin with the slash / character. To do this, set the **CDPATH** variable in your *.kshrc* file.

The **ksh** provides an option to **cd** that allows you to return quickly to your previous working directory. For example, if you are in the */u/sue/bin* directory and you enter:

```
cd /u/sue
```

you can return to your previous directory, */u/sue/bin,* by entering **cd -** . From this directory, you can enter **cd -** again to return to the */u/sue* directory.

ksh provides a means for changing to a directory with a pathname that is slightly different from your current working directory, without having to enter the full path to it. To do this, use the following format:

```
cd old new
```

where ***old*** is the part of the pathname that you want to change and ***new*** is what you want to change it to. For example, if you are in */usr/spool/mail* and you want to change to */usr/bin/mail* , enter:

```
cd spool bin
```

The **ksh** also behaves intelligently when working out where you are in the file system. **ksh** has a built-in version of the **pwd** command which recognizes symbolic links to other directories. Suppose, for example, you are working in a directory called */usr/fred/work/here/now*. Your home directory is */usr/me*. Because */usr/fred/work/here/now* is hard to type, you have created a *link* called *w* from your home directory to the work directory, using the command

```
ln -s /usr/fred/work/here/now w
```

And to get from your home directory to */usr/fred/work/here/now* you simply type

```
cd w
```

ksh's **pwd** command understands links; when you type

pwd

ksh responds with

```
/usr/me/w
```

This is in contrast to the program **/bin/pwd** which responds with

```
/usr/fred/work/here/now
```

You can return to your home directory (*/usr/me*) from */usr/fred/work/here/now* by entering either

cd ..

or

cd /usr/me

because **ksh**'s **cd** command recognizes the symbolic link */usr/me/w* as being the route by which you entered the work directory.

If you want to turn this behaviour off, forcing **pwd** and **cd** to ignore symbolic links, you can either use the commands with the **-P** argument, as in

cd -P ..

or you can add the following lines to your *.kshrc* file:

```
alias cd='cd -P'
alias pwd='pwd -P'
```

Using job control

The job control feature of **ksh** allows you to manipulate jobs running in the foreground and background. A job is a program that a user has started but which requires no user input while it is running; examples include database sorting and program compilation. With job control, you can stop and restart programs, and move them between the foreground and the background. This is particularly useful for dealing with long, non interactive tasks, such as formatting a long document or making a large program. You can also specify the priority of a job, using **nice**, and permit jobs to continue running after you log out, using **nohup**.

Like **sh** and **csh**, **ksh** runs commands in the foreground by default; the shell waits for the current command to finish executing before displaying the prompt. You run a command in the background by adding an ampersand (&) character to the end of the command before pressing ⟨Return⟩. Before displaying the prompt, the shell displays the status of any completed background jobs.

When job control is active and you run a command in the background, **ksh** displays both the number of the job in square brackets, `[ ]`, and the process ID number (PID).

Job control is active by default. If job control is disabled, **ksh** displays the following error message when you try to manipulate foreground and background jobs:

```
no job control
```

To activate job control, add the following line to your environment file (*.kshrc*):

"set -o monitor"

To use job control, Suspend must be set (usually ⟨Ctrl⟩-**z**). To set it, add the following line to your **$HOME**/*.profile* file:

```
"stty susp '^Z'"
```

Referring to jobs

When you use **ksh** as your login shell, you can refer to jobs in several different ways:

PID	refers to the job by the process ID number.
%number	refers to the job by the job number that **ksh** displays in square brackets.
%string	refers to the job whose command begins with *string*.
%?string	refers to the job whose command contains *string*.
%+ or %%	refers to the current job.
%-	refers to the previous job.

For example, if you enter the command **sleep 30&**, and **ksh** displays:

```
[1]   3456
```

you can refer to this background job in various ways, including the following:

3456
%1
%sle
%?ee

Using the ksh job control commands

The **ksh** includes the following built-in commands for job control. These commands take a PID, job name, or number as the argument:

Table 10-7 Job control commands

Command	Description
bg	starts the specified stopped job in the background.
fg	moves the specified background job to the foreground.
jobs	displays status information about current jobs.
kill	terminates the specified background job.
wait	tells **ksh** to wait for all or a specific background process to complete.

Running jobs in the background

To run a job in the background, enter the command name, followed by an ampersand (&) character and press ⟨Return⟩.

For example, when you enter

```
    sleep 30&
```

ksh starts the job in the background and displays a message like the following:

```
    [1]   3456
```

When the job finishes, **ksh** displays a message like this:

```
    [1] + Done              sleep 30&
```

Moving background jobs to foreground

To move a job that is running in the background to the foreground, enter **fg** followed by the PID, command name, or job number.

For example, move the *sleep* background process to the foreground by entering:

```
    fg %?ee
```

Once the job is in the foreground, **ksh** waits for it to complete before displaying a prompt.

Moving foreground jobs to background

To move a foreground job to the background, press the suspend key. (This is usually -Z, but can be altered; if your system does not respond to -Z check the **stty susp** line in your *.profile* file.) The **ksh** stops the job and displays a message like the following:

```
[1] + Stopped          sleep 30&
```

The **ksh** displays your prompt. At the prompt, enter **bg** followed by the PID, job number, or name. For example, move the **sleep** process to the background by entering:

bg %?ee

The **ksh** restarts the suspended job, displays a message like the following, and runs the job in the background:

```
[1]  sleep 30&
```

Displaying information about jobs

Use the **jobs** command to display status information about the jobs that **ksh** is currently running. To display information about all active jobs, enter **jobs** at the command line. The **ksh** displays status information in the following format:

```
[2] + Running          sleep 40&
[1] - Stopped          sleep 30&
```

The **ksh** displays a plus (+) character after the job number of the current job, a minus (-) after the previous job.

Use the **-l** option to display the PID after the job number. For example, if you enter

```
jobs -l
```

ksh displays information like the following:

```
[1] + 23909  Stopped          sleep 30&
```

You can limit the display to PID only by using the **-p** option to **jobs**.

Terminating a background job

To terminate a process, use the **kill**(C) command. The syntax for **kill** is:

kill [-*signal*] *job*

where ***signal*** is an optional signal number or name and ***job*** is the job name, number, or PID of the job that you want to terminate.

To display a complete list of signal numbers and names, enter **kill -l**. Signals are used by UNIX as a way of communicating between processes. The **kill**(C) command is simply a program that can be used to send signals to other processes, in order to suspend or kill them.

If you do not specify ***signal***, **ksh** sends the TERM (terminate) signal to the specified job. See the previous section "Referring to jobs" in this chapter for more information on referring to jobs by job name, number, and PID .

When you use **kill ksh** displays a message like the following and terminates the job:

```
[1] + Terminated        sleep 30&
```

External job control facilities

In addition to the job control facilities built into **ksh** it is worth remembering two other programs at this point; **nice** and **nohup**. While **nice** and **nohup** are not part of the shell, these utilities complement the job control features built into **ksh**.

If a large number of users are logged onto the system or a large number of jobs are running in the background, system performance may be significantly degraded. To reduce the likelihood of this happening, it is possible to fine tune the resource allocation of a job. **nice** can be used to reduce the amount of time allocated to a job by the processor. For further information on using **nice**, see **nice**(C).

Occasionally it may be necessary to run an extremely long job in the background which needs to continue even after you have logged out. When you log out of the Korn shell, a HUP signal is sent to all **ksh**'s child processes. The HUP (hangup) signal causes a process receiving it to terminate.

To enable a program to continue running after you log out, it is necessary to run the program using **nohup**. (**nohup** traps the HUP signal, allowing your program to run on without interruption.) For example,

```
nohup very_long_job &
```

causes **very_long_job** to be run as a background process that continues even after the parent process is logged out. For further information, see **nohup**(C).

Interpretation of commands

Like **sh**, the Korn shell is a fully functional command language interpreter. The language provided by **ksh** is an extended version of the **sh** language. Among the additional language features **ksh** provides are:

- a menu selection primitive (select) which provides a means of querying the user via menus.
- built in integer arithmetic; **ksh** can perform integer arithmetic using **ksh** variables and constants.
- string operators that enable **ksh** to convert strings to upper- or lower-case and return substrings.
- array handling facilities so that **ksh** can operate upon one dimensional arrays of strings or numbers.
- function definitions; **ksh** recognizes variables local to a function; consequently recursive functions can be defined and it is easier to write well structured programs.

In general, most existing **sh** scripts run under **ksh** without alteration. This is because **ksh** is effectively a superset of **sh**, and duplicates almost all the functionality of the earlier shell, while adding additional features. To understand how **ksh** works, it is necessary to start by looking at how commands are carried out.

ksh first reads a command from the standard input, then executes it. Compound commands such as a **for ... in ... do ... done** loop are expanded and executed at every iteration; functions are expanded and executed each time they are called. The process of expansion consists of substituting aliases, expanding quotes, and interpolating variables. Consequently, the command which **ksh** ultimately executes may not resemble the line which **ksh** is fed as input.

For example, after the earlier commands

```
alias whereis='find / -name $1 -print 2> /dev/null'
set fubar='/etc/motd'
```

are executed, the command

```
whereis fubar
```

is interpreted as

```
find / -name=/etc/motd -print 2> /dev/null
```

Alias expansion

The **ksh** alias mechanism is similar to that of the C shell in that it carries out transformations upon commands immediately after they are input. The command is checked against a list of aliases known to **ksh**, and if it is recognized the corresponding meaning of the alias is substituted. This feature is used to provide synonyms for commands and command sequences to improve the usability of the shell.

To define an alias, enter the command **alias**, followed by the name of the alias to use and then the definition of the alias. For example:

```
alias -x look='vi $1'
```

assigns the alias **look** to the command **vi $1**. (Note that the **-x** flag, short for export, marks the alias automatically so that it is exported to any programs run from the shell.) When the command

```
look my_file
```

is typed, **ksh** expands the alias **look** to **vi $1**. (The process of replacing a reference to an alias with its value is known as "expansion".) **$1** is the first argument of the command, as with **sh**, so *my_file*, which is the first argument on the current line, is substituted for ***$1***. Consequently, the command which is actually executed is

```
vi my_file
```

NOTE If you enter a ⟨Tab⟩ or ⟨Space⟩ before or after the "=" in the alias assignment **ksh** may misinterpret your definition.

In general, aliases take the form

alias [-x] ***name=value***

where ***name*** is the alias to which ***value*** is assigned.

If ***value*** is enclosed in single quotes, it is expanded only when **ksh** processes a reference to it; if it is enclosed in double quotes, **ksh** expands it when it first processes the alias command.

For example, in the command

```
alias -x edit='vi $1'
```

$1 is expanded when the alias is called, so that

```
edit foo.bar
```

will result in

```
vi foo.bar
```

being executed; but the command

```
alias -x edit="vi $1"
```

is expanded when the alias is defined, so that **$1** is not interpreted in context when **edit** is used.

Aliases can contain any valid **ksh** command structure. It is safe to use the name of an alias inside its value. For example, in the definition

```
alias -x ls='ls -al'
```

the **ls** within the quotes is not replaced again by

'ls -al'.

If the value of an alias ends with a space or tab character, **ksh** automatically checks the next command word it encounters for alias substitution. For example, the following dialogue sets up an alias for **print** that checks its argument for aliases:

```
1% alias -x hello='print '
2% alias -x world='hello world'
3% hello world
hello world
4%
```

While this one doesn't check:

```
1% alias -x hello='print'
2% alias -x world='hello world'
3% hello world
world
4%
```

There are two parameters to **alias**:

1. **-x** in an alias definition causes the alias to be exported to sub shells; otherwise aliases are specific to the shell in which they are defined. If **alias -x** is entered without a definition, a list of all the current exportable alias definitions is sent to the standard output.
2. **-t** in an alias definition flags the alias as a "tracked" definition. A tracked alias is one in which the full pathname corresponding to the program given in the alias definition is substituted for the name in the definition; for example:

   alias -t mv

   would result in **mv** having the value **/bin/mv**. Use of a tracked alias reduces the amount of time it takes **ksh** to find and execute a program. The value of a tracked alias becomes undefined whenever the **PATH** variable is reset, but the alias remains defined as being tracked. The next time the alias is referenced, its value is redefined.

If the **-t** flag is used without an alias definition, a list of all existing tracked aliases is sent to the standard output.

Quote expansion and ksh

In addition to expanding aliases encountered in its input, **ksh** operates upon tokens encountered between quotation marks. (A token is the smallest unit of input recognized by **ksh** as being either a command or an item of data.)

If a command is enclosed between backquotes (**` ... `**) or within brackets preceded by a "**$**" symbol (**$(...)**), the bracketed command is replaced by the *output* of that command. For example:

```
where_am_i=`pwd`
```

or

```
where_am_i=$(pwd)
```

assigns the value returned by **pwd** to the variable **where_am_i**, but

```
where_am_i='pwd'
```

assigns the string "pwd" to the variable **where_am_i**.

Ordinary quotes (single or double) do not result in command substitution being carried out in this way. However, **ksh** does not interpret commands enclosed in quotes in exactly the same way as unquoted commands.

ksh obeys the same rules for interpreting quoted text as **sh**. Literal (single) quotes, (**' ... '**) remove the special meaning of all enclosed characters. A single quote can never appear within a single quoted string because it is interpreted as the end of the string.

Double quoted (grouping) quotes remove the special meaning of all enclosed characters, except "**$**", "**`**", "**"**" and "\".

Within double quotes, when **ksh** encounters one of the above characters preceded by a backslash, the backslash is discarded and the character is interpreted with its literal meaning. When not preceded by a backslash, the special characters are interpreted as follows:

$	Parameter expansion
$(...)	Command substitution (new style, not shared with **sh**).
"**`**"	Old style command substitution
"**"**"	End of current string
" "	Escape character

Parentheses

In addition to the **$(...)** construct, there are a few other parenthesized forms that are unique to **ksh**.

The **${ ... }** form is used to indicate that parameter expansion should be applied to the parameters within the braces. For full details of this process, refer to **ksh**(C) ; briefly, the parameters in the braces are treated as variables, and their values substituted.

The **((...))** form is used to denote an arithmetic evaluation. **ksh** is capable of evaluating arithmetic statements of the form **let " ... "**. The double bracket notation is a shorthand for this, and text contained between double brackets is evaluated arithmetically.

Single square brackets **[...]** are used to denote array subscripts. **ksh**, unlike the other shells, is capable of handling one dimensional arrays of variables. The text within the square brackets immediately after the name of the array is evaluated and used as a subscript: that is, an index that indicates which elements of the array are being referred to.

Double square brackets **[[...]]** are used to contain test expressions for some conditional statements. The expression contained between double square brackets is evaluated, and a zero exit status is returned if the expression is true. This notation is unlikely to be encountered unless you are writing or maintaining shell programs.

Tilde expansion

Tilde Expansion is a feature of **ksh** which is not shared by the other shells. When a word beginning with a tilde ("~") is encountered in **ksh**'s input after alias substitution, the word is checked as far as the first "/" to see if it matches:

"~"	(by itself) is replaced by the value of **$HOME**.
"~+"	this is replaced by the string **$PWD**, which is then expanded.
"~-"	this is replaced by **$OLDPWD**, which is then expanded.
"~"	when followed by user login name, this is replaced by the home directory of the user.
"~"	followed by anything else is left unchanged.

Programming the Korn shell

Programs written in the Korn Shell language are known as shell scripts. They can be written using any of the standard text editors. To execute a shell script, all that is necessary is to use **chmod +x** to make the file executable, then type the name of the script file at the shell prompt (followed by any necessary arguments). **ksh** then runs the script.

You can also run a script by running **ksh** with the name of the script as an argument; this causes a second copy of **ksh** to execute and run the script.

At its simplest, a **ksh** script might consist of a list of program names and their arguments, each located on a new line or separated from the preceding and following commands by a semicolon. Such a list, when read by **ksh**, is interpreted one command at a time. Each command is checked for aliases, which are expanded as necessary, then all required quote and variable substitutions are carried out. Finally, **ksh** forks and executes the command, then advances to the next line. When there are no more lines to execute, **ksh** terminates the script and closes the file.

Arguments, parameters, and variables

As with **sh**, it is possible to pass arguments to a shell script via the command line. Arguments to the script are stored as positional parameters. Positional parameter "0" is set to the name of the script, and parameters numbered sequentially from "1" onwards are set to the value of the arguments.

Parameters are accessed within the script using the following notation:

$*n*	Refers to the *n*'th parameter. For example, the third parameter is referred to by **$3**. Braces are required around the parameter number for values of ***n*** greater than 9. (Unlike **sh**, **ksh** does not place an upper limit on the number of parameters which are permitted.)
$#	The number of positional parameters, not including 0.
"$*"	A single argument consisting of all the positional parameters except 0. (Note that quotes are required to prevent parameters with embedded white space from being split into separate arguments and null arguments from being removed.)

The **set** command can be used to assign new values to or unset the values of the positional parameters.

In addition to the arguments provided when the script is first executed, **ksh** can access environment parameters and its own internal parameters in the course of execution. Such named parameters are referred to as variables. Some of the parameters are preconfigured and contain information relating to the state of the shell; others are defined by the script and can be used for temporary storage during execution.

ksh lets you define as many variables as you need. User defined parameter names are preceded by a" $ " symbol to identify them to **ksh**.

By default, ksh treats all parameters as strings of text. However, unlike **sh**, **ksh** can distinguish different types of variable. You can specify that a variable is to be treated as an integer number by using the **typeset -i** option to tag it with the integer attribute. Whenever **ksh** expands a parameter tagged with the **-i** attribute it treats it as an integer rather than a string. Indeed, there are a number of attributes available which control the manner in which variables are treated by the shell. Any variable can be tagged with any attribute by use of the **typeset** command; but some of the attributes are mutually exclusive. For example, the **-u** (uppercase) attribute and the **-l** (lower case) attribute cancel one another; only the most recently applied takes effect.

The following attributes are available:

-H This flag provides UNIX system to hostname file mapping on non-UNIX system machines.

-L Left justify and remove leading blanks from ***value***. If ***n*** is non zero it defines the width of the field: otherwise it is determined by the width of the value of the first assignment. When the parameter is assigned to, it is filled on the right with blanks or truncated, if necessary, to fit into the field. Leading zeros are removed if the **-Z** flag is also set. The **-R** flag is turned off.

-R Right justify and fill with leading blanks. If ***n*** is non zero it defines the width of the field, otherwise it is determined by the width of the value of the first assignment. The field is left filled with blanks or truncated from the end if the parameter is reassigned. The **L** flag is turned off.

-Z Right justify and fill with leading zeros if the first non blank character is a digit and the **-L** flag has not been set. If ***n*** is non zero it defines the width of the field, otherwise it is determined by the width of the value of first assignment.

-f The names refer to function names rather than parameter names. No assignments can be made and the only other valid flags are **-t**, **-u** and **-x**. The flag **-t** turns on execution tracing for this function. The flag **-u** causes this function to be marked as undefined. The **FPATH** variable is searched to find the function definition when the function is referenced. The flag **-x** allows the function definition to remain in effect across shell procedures invoked by name.

-i Parameter is an integer. This makes arithmetic faster. If *n* is non zero it defines the output arithmetic base; otherwise the first assignment determines the output base.

-l All upper-case characters converted to lower case. The upper-case flag, **-u**, is turned off.

-r The given ***names*** are marked readonly and these names cannot be changed by subsequent assignment.

-t Tags the named parameters. Tags are user definable and have no special meaning to the shell.

-u All lower-case characters are converted to upper-case characters. The lower-case flag, **-l** is turned off.

-x The given ***name***s are marked for automatic export to the environment of subsequently executed commands.

Using + rather than - causes these flags to be turned off. If no ***name*** arguments are given but flags are specified, a list of names (and optionally the values) of the parameters which have these flags set is printed. (Using + rather than - keeps the values from being printed.) If no ***name***s and flags are given, the names and attributes of all parameters are printed.

Built-in commands

Built-in commands (sometimes referred to as internal commands) are recognized as keywords by **ksh**, which processes them itself. It is not necessary for **ksh** to fork and execute another process to carry out an internal command; consequently scripts consisting only of internal commands run faster than those which frequently make use of external commands. Because internal commands are part of **ksh**, they can also be used to alter the current environment.

There are two categories of built in command in **ksh**. There are simple commands, like **typeset** or **export**, and flow control commands like **for** and **while.** The flow control commands tend to be more complex because they are used to control the execution of the other commands.

All internal commands return a value when executed, just like any other command. The return value **FALSE** (1) is returned if a command is specified for which any of the following are true:

- the wrong number of arguments was specified.
- an invalid argument was specified (for example, a string argument when a numeric one was expected).
- an invalid variable was assigned.
- an invalid alias name was specified.
- an invalid I/O redirection was requested.
- an expansion for an unset parameter is specified and **nounset** is on.
- an invalid option is selected.

If a command succeeds, a non false value is returned, the type of which depends on the command in question.

The following reserved words are recognized as the first word of an internal command when not quoted:

if	**then**	**else**
elif	**fi**	**case**
esac	**for**	**while**
until	**do**	**done**
{ }	**function**	**select**
time	**[[ ]]**	

All the above listed internal commands behave like their equivalents in **sh,** except for **select,** which is described below.

Comments are denoted by a word beginning with a " # ". The presence of this symbol causes that word and all the following characters up to a new line to be ignored.

Differences from the Bourne shell

As noted previously, the syntax recognized by **ksh** is a superset of the standard **sh** language. All the normal shell program constructs are available. In addition to these the job control commands, alias expansions and variable attributes are available. **ksh** provides a few more programming features which are not available under **sh**. These are described below.

The additional constructs are:

- functions
- substring expansions
- the **select** command
- pattern matching
- debugging support

Functions

Functions provide the **ksh** with a degree of structure which enables complex programs to be written, debugged and maintained more easily than in the standard shell. A set of tasks which are carried out repeatedly in the course of a program can be encapsulated within a function. The function must be defined (declared) in the main program before it can be used. The act of declaring a function does not actually cause the actions defined in the function to be carried out; but when the function name is called from the main program the function definition is recalled and the commands contained in it are executed.

A function differs from a dot script in that variables can be defined which are local to the function, and positional parameters are saved and restored prior to and after function execution. Functions execute faster than dot scripts because they are read once only, when they are defined, rather than every time that they are invoked. Because functions execute within the current environment, they can share variables with the script from which they are invoked. They can also invoke other functions.

Local variables, which are only accessible within the body of a function, can be created by using typeset within the function. Local variables can possess the same name as a variable in the main script, but assigning a value to a local variable has no effect on the variable in the calling script. However, if a new value is assigned to a variable declared in the main script, the change is retained when the function returns.

Functions can have parameters. When calling a function, specify any parameters to it as you would the parameters to a shell script or ordinary function. Positional parameter 0, as with a script, is set to the name of the function.

After expanding the current command line, **ksh** checks the current command to see if it is an internal command, then a function, then an external command. Consequently if a function possesses the same name as an external command, the external command cannot be executed; and if a user defined function possesses the same name as an internal command, it *cannot* be carried out.

To declare a function, the command **function** is required and the body of the function is enclosed in braces:

```
function equal_args     # Returns true (0) if $1 = $2
{
        if (( $1 == $2  ))
                then return 0
        else
                print error
                return 1
        fi
}
```

Note that **$1** and **$2** in this example refer to the positional parameters supplied to the function.

To display the definition of one or more functions, use the **functions** alias. To remove a function from the shell, use **unset -f** followed by the name of the function. Because function definitions are not inherited by sub-shells, any functions which are required in all sub-shells created by **ksh** should be defined in the *.kshrc* file.

It is possible to define autoload functions which are loaded by the shell only when they are first called. Such functions are declared in the shell script using the **autoload** alias. When **ksh** encounters the name of an autoload function, it searches the path defined in **FPATH** for a file with the same name as the function. **ksh** then reads this file into the environment before it executes the function. This confers the advantage that **ksh** does not read the function definitions unless and until the function is called within a program.

Because alias expansion occurs before commands are interpreted, it is possible to use an alias in conjunction with a function to redefine an internal **ksh** command. First, a function to carry out the desired action is defined. It should have a different name from the internal command it is to replace, otherwise the internal command is invoked in preference to the function. Secondly, an alias with the *same* name as the internal command is defined, its value being the name of the function. Because the alias is evaluated before any commands are carried out, all instances of the internal command name are expanded into the name of the function to replace that command.

For example, using a combination of alias and function definitions it is possible to redefine the **cd** command to behave in a way more familiar to users of MS-DOS.

```
_cd() {
        case $# in
        0)      pwd                  ;;
        *)      cd "$@" && PS1="$PWD> " ;;
        esac
}

alias   cd=_cd
```

Under UNIX, the command **cd** either changes to the specified directory (if an argument is given), or changes to the users home directory (if no arguments are given). This may confuse users who are converting from MS-DOS; under MS-DOS, if no arguments are used, **cd** echoes the present working directory (that is, it behaves like the UNIX command **pwd**).

Firstly, the function **_cd()** is defined. If **_cd()** is called without arguments, it executes **pwd**. If it is called with one or more arguments it **cd**'s to the directory given by the argument, and if successful sets the current command prompt to the value of the present working directory (in a manner similar to the MS-DOS **prompt pg** command).

Secondly, the **cd** command is redefined as an alias to **_cd()**. Whenever the command **cd** is entered, the function **_cd()** will be executed instead.

Substring parameter expansion

ksh provides a variety of string manipulation facilities, which are applied to variables at the parameter expansion stage of command execution. Using the substring parameters listed below, it is possible to chop and change variables and conduct pattern matching searches upon data held by the program. Substring parameter expansion can be applied to any parameter. The normal **${ ... }** syntax is used to indicate that the parameters in braces are to be expanded; however a number of flags may be included, which indicate the way in which the parameters are to be treated.

The substring expansions available to **ksh** programs include the following:

${#*parameter*} If ***parameter*** is * or @, the number of positional parameters is substituted. Otherwise, the length of the value of the parameter is substituted.

${#*identifier*[*]} The number of elements in the array identifier is substituted.

${*parameter:-word*} If ***parameter*** is set and is non null then substitute its value; otherwise substitute ***word***.

${*parameter:=word*} If ***parameter*** is not set or is null then set it to ***word***; the value of the parameter is then substituted. Positional parameters may not be assigned to in this way.

${*parameter:?word*} If ***parameter*** is set and is non null then substitute its value; otherwise, print ***word*** and exit from the shell. If ***word*** is omitted then a standard message is printed.

${*parameter:+word*} If ***parameter*** is set and is non null then substitute ***word***; otherwise substitute nothing.

${*parameter#pattern*}

${*parameter##pattern*} If the shell pattern matches the beginning of the value of ***parameter***, then the value of this substitution is the value of the parameter with the matched portion deleted; otherwise the value of this parameter is substituted. In the first form the smallest matching pattern is deleted and in the second form the largest matching pattern is deleted.

${*parameter%pattern*}

${*parameter%%pattern*} If the shell pattern matches the end of the value of ***parameter***, then the value of this substitution is the value of the ***parameter*** with the matched part deleted; otherwise substitute the value of ***parameter***. In the first form the smallest matching pattern is deleted and in the second form the largest matching pattern is deleted.

In the above, ***word*** is not evaluated unless it is to be used as the substituted string, so that, in the following example, **pwd** is executed only if **d** is not set or is null:

```
        echo ${d:-$(pwd)}
```

If the colon (:) is omitted from the above expressions, then the shell only checks whether ***parameter*** is set or not.

The select command

The **select** command is a special purpose command intended to make it easier to write scripts which require user input. The syntax of **select** is as follows:

> **select** ***identifier*** **[in *word* . . .]**
> **do *compound list***
> **done**

ksh performs command and parameter substitution, word splitting, pathname expansion and quote removal on each ***word,*** to produce a list of items before it processes the **do** C compound list command. If **in*word*** is not specified, **ksh** substitutes the positional parameters (starting at 1) as the list of items.

The **select** command is executed as follows:

- The list of items is displayed in one or more columns on the standard error. Each item is preceded by a number. Below the items, **ksh** displays the PS3 prompt. The number of lines and columns used are derived from the values of the **LINES** and **COLUMNS** variables.
- **ksh** reads a line from the standard input. If the line is empty, **ksh** redisplays the list of items and the PS3 prompt until a non empty line is read. If the line consists of the number of one of the displayed items, **ksh** sets the value of the variable ***identifier*** to the item corresponding to the number.
- ksh saves the contents of the non empty selection line in the variable **REPLY.**
- **ksh** executes ***compound list*** for each selection until a **break, return** or **exit** command is encountered in ***compound list.***
- **select** terminates when it encounters an End of file condition.

The following example shows a simple menu and returns the results of the selection:

```
one='one'
two='two'
three='three'
four='four'
five='last'
PS3='Please enter a number:'
select choice in $one $two $three $four $five
do print 'You entered number ' $REPLY
   print 'and selected item ' $choice;
   break;
done
```

Note that the menu appears on the standard error. Consequently it is important that the standard error should be redirected to the same terminal as the standard input if **select** is to be used interactively.

Pattern matching

The **ksh** pattern matching facilities are extensive, and differ from those provided by **grep, ed** or other programs. Patterns can be used in pathnames, **case** pattern matching, pattern matching within [[...]], and substring expansion.

The following pattern characters are recognized by **ksh** and are expanded:

"*"	Matches any string, including the null string.
"?"	Matches any single character.
"[...]"	Matches any one of the enclosed characters. A pair of characters separated by "-" matches any character lexically between the pair, inclusive. If the first character following the opening "[" is a "!" then any character not enclosed is matched. A "-" can be included in the character set by putting it as the first or last character.

A *pattern list* is a list of one or more patterns separated from each other with a \|. Composite patterns can be formed with one or more of the following:

?(*pattern list*)	Optionally matches any one of the given patterns.
***(*pattern list*)**	Matches zero or more occurrences of the given patterns.
+(*pattern list*)	Matches one or more occurrences of the given patterns.
@(*pattern list*)	Matches exactly one of the given patterns.
!(*pattern list*)	Matches anything, except one of the given patterns.

Debugging support

ksh provides facilities that make it easier to detect problems with scripts. When **ksh** detects a syntax error while reading a script it displays the script name, the line number at which it became aware of the error, and the probable cause of the error. When **ksh** detects a runtime error while executing a script it displays the function or script name, line number (n square brackets) and error. Line numbers are relative to the beginning of the script file when executing a script and relative to the first line of a function when executing a function.

In addition to these basic features, **ksh** provides a number of other debugging aids. You can check the syntax of a script without executing it, by running **ksh** with the **noexec** option; or you can make **ksh** display its input by setting the **verbose** option.

You can also make **ksh** display an execution trace as it executes your script. The execution trace is a listing of each command that **ksh** encounters, as it is expanded but before it is executed. **ksh** can be made to produce a trace output by either turning on the **xtrace** option with **set -x** or **set -o xtrace** inside the script, or by invoking **ksh** with the **xtrace** parameter. To trace a function, use **typeset** with the **-ft** options.

The PS4 prompt is displayed in front of each line during the execution trace. To follow the line numbering within the script, make sure that the **PS4** prompt contains the **LINENO** variable. If the appropriate variables are included in the prompt, you can trace their values as execution proceeds.

Finally, **ksh** provides the **DEBUG** trap. To set the **DEBUG** trap, use the command

trap [*action*] debug

Where ***action*** is a command to execute after every simple command. For example, you can specify that the action to take is something like

```
trap print $firstvar $secondvar DEBUG
```

in which case the value of each variable is printed after each line is executed, until the trap is removed.

Chapter 11

Manipulating text with sed

This chapter describes the stream editor, **sed**(C), that allows you to perform large-scale, noninteractive editing tasks. The **sed** editor is useful for working with large files or running complicated sequences of editing commands on a file or group of files.

Although you can perform many of the same tasks with **grep**, **sort**, and the variants of **diff**, **sed** offers an added facility for the processing of complicated changes to large files, or many files at once. **sed** is very handy for large batch editing jobs, however, you can perform many of the same tasks with **ed** scripts.

The **sed** program is a noninteractive editor which is especially useful when the files to be edited are either too large, or the sequence of editing commands too complex, to execute interactively. **sed** works on only a few lines of input at a time and does not use temporary files, so the only limit on the size of the files you can process is that both the input and output fit simultaneously on your disk. You can apply multiple "global" editing functions to your text in one pass. You can create complicated editing scripts and submit them to **sed** as a command file. Therefore, you can save yourself considerable retyping and the possibility of making errors. You can also save and reuse **sed** command files that perform editing operations that you repeat frequently.

Processing files with **sed** command files is more efficient than using **ed**, even if you prepare a prewritten script. Note, however, that **sed** lacks relative addressing becauses it processes a file one line at a time. Also, note that **sed** gives you no immediate verification that a command has altered your text in the way you actually intended. For this reason, you should check your output carefully.

The **sed** program is derived from **ed**, although there are considerable differences between the two, resulting from the different characteristics of interactive and batch operation. However, there is a striking resemblance in the class of regular expressions that each program recognizes; the code for matching patterns is nearly identical for **ed** and **sed.**

Using sed

By default, **sed** copies the standard input to the standard output, performing one or more editing commands on each line before writing it to the output. Typically, you need to specify the file or files that you are processing, along with the name of the command file that contains your editing script. For example:

sed -f ***script filename***

The flags, such as **-f** are optional. The **-f** flag in this example tells **sed** to take the next argument as a filename. (This file must contain editing commands, one to a line.)

The general format of a **sed** editing command is:

address1,address2 function arguments

In any command, you can omit one or both addresses. A function is always required, but an argument is optional for some functions. Any number of blanks or tabs can separate the addresses from the function, and tab characters and spaces at the beginning of lines are ignored.

Three flags are recognized on the command line:

-n	Directs **sed** to copy only those lines specified by **p** functions or **p** flags after s functions.
-e	Indicates that the next argument is an editing command.
-f	Indicates that the next argument is the name of the file which contains editing commands, typed one to a line.

sed commands are applied one at a time, generally in the order they appear, unless you change this order with one of the "flow-of-control" functions discussed below. **sed** works in two phases, compiling the editing commands in the order they are given, then executing the commands one by one to each line of the input file.

The input to each command is the output of all preceding commands. Even if you change this default order of applying commands with one of the two flow-of-control commands, **t** and **b**, the input line to any command is still the output of any previously applied command.

You should also note that the range of pattern match is normally one line of input text. This range is called the "pattern space." More than one line can be read into the pattern space by using the **N** command described below in "Multiple input-line functions."

The rest of this section discusses the principles of **sed** addressing, followed by a description of **sed** functions. All the examples here are based on the following lines from Samuel Taylor Coleridge's poem, "Kubla Khan":

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

For example, the following command quits after copying the first two lines of the input:

```
2q
```

Using the sample text, the result is:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

Addresses

The following rules apply to addressing in **sed**. There are two ways to select the lines in the input file to which editing commands are to be applied: with line numbers or with "context addresses."

Context addresses correspond to regular expressions. You can control the application of a group of commands by one address or an address pair, by grouping the commands with curly braces ({ }). You can specify 0, 1, or 2 addresses, depending on the command. The maximum number of addresses possible for each command is indicated.

A line number is a decimal integer. As each line is read from the input file, a line number counter is incremented. A line number address matches the input line, causing the internal counter to equal the address line number. The counter runs cumulatively through multiple input files. It is not reset when a new input file is opened. A special case is the dollar sign character ($) which matches the last line of the last input file.

Context addresses are enclosed in slashes (/). They include all the regular expressions common to both **ed** and **sed:**

- An ordinary character is a regular expression and matches itself.
- A caret (^) at the beginning of a regular expression matches the null character at the beginning of a line.
- A dollar sign ($) at the end of a regular expression matches the null character at the end of a line.
- The characters "\n" match an embedded newline character, but not the newline at the end of a pattern space.
- A period (.) matches any character except the terminal newline of the pattern space.
- A regular expression followed by a star (*) matches any number, including 0, of adjacent occurrences of regular expressions.
- A string of characters in square brackets ([ ]) matches any character in the string, and no others. If, however, the first character of the string is a caret (^), the regular expression matches any character except the characters in the string and the terminal newline of the pattern space.
- A concatenation of regular expressions is one that matches a particular concatenation of strings.
- A regular expression between the sequences "\(" and "\)" is identical in effect to itself, but has side-effects with the **s** command. (Note the following specification.)
- The expression "\d" means the same string of characters matched by an expression enclosed in "\(" and "\)" earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of \(, counting from the left. For example, the expression ^\(.*\)\1 matches a line beginning with two repeated occurrences of the same string.
- The null regular expression standing alone is equivalent to the last regular expression compiled.

For a context address to "match" the input, the whole pattern within the address must match some portion of the pattern space. If you want to use one of the special characters literally, that is, to match an occurrence of itself in the input file, precede the character with a backslash (\) in the command.

Each **sed** command can have 0, 1, or 2 addresses. The maximum number of allowed addresses is included. A command with no addresses specified is applied to every line in the input. If a command has one address, it is applied to all lines that match that address. On the other hand, if two addresses are specified, the command is applied to the first line that matches the first address, and to all subsequent lines until and including the first subsequent line that matches the second address. An attempt is made to match the first address on subsequent lines, and the process is repeated. Two addresses are separated by a comma. Here are some examples:

/an/	matches lines 1, 3, 4 in our sample text
/an.*an/	matches line 1
/^an/	matches no lines
/./	matches all lines
/r*an/	matches lines 1,3, 4 (number = zero!)

Functions

All **sed** functions are named by a single character. They are of the following types:

- whole-line oriented functions that add, delete, and change whole text lines.
- substitute functions that search and substitute regular expressions within a line.
- input-output functions that read and write lines and/or files.
- multiple input-line functions that match patterns that extend across line boundaries.
- hold and get functions that save and retrieve input text for later use.
- flow-of-control functions that control the order of application of functions.
- miscellaneous functions.

Whole-line oriented functions

d Deletes from the file all lines matched by its addresses. No further commands are executed on a deleted line. As soon as the **d** function is executed, a newline is read from the input, and the list of editing commands is restarted from the beginning on the newline. The maximum number of addresses is two.

n Reads and replaces the current line from the input, writing the current line to the output if specified. The list of editing commands continues following the **n** command. The maximum number of addresses is two.

- **a** Causes the text to be written to the output after the line matched by its address. The **a** command is inherently multi-line and must appear at the end of a line. The text can contain any number of lines. The interior newlines must be hidden by a backslash character (\) immediately preceding each newline. The text argument is terminated by the first unhidden newline, the first one not immediately preceded by backslash. Once an **a** function executes successfully, the text is written to the output regardless of what later commands do to the line that triggered it, even if the line is subsequently deleted. The text is not scanned for address matches, and no editing commands are attempted on it, nor does it cause any change in the line number counter. Only one address is possible.
- **i** When followed by a text argument, **i** functions the same as **a**, except that the text is written to the output before the matched line. It has only one possible address.
- **c** The **c** function deletes the lines selected by its addresses, and replaces them with the lines in the text. Like the **a** and **i** commands, **c** must be followed by a newline hidden with a backslash; interior newlines in the text must be hidden by backslashes. The **c** command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of the text is written to the output, not one copy per line deleted. As in the case of **a** and **i**, the text is not scanned for address matches, and no editing commands are attempted on it. It does not change the line number counter. After a line has been deleted by a **c** function, no further commands are attempted on it. If text is appended after a line by **a** or **r** functions, and the line is subsequently changed, the text inserted by the **c** function is placed before the text of the **a** or **r** functions.

Note that when you insert text in the output with these functions, leading blanks and tabs disappear in all **sed** commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash does not appear in the output.

For example, apply the following list of editing commands to our standard input:

```
n
a\
XXXX
d
```

The output is:

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, you get the same effect using either of the two following command lists:

```
n
i\
XXXX
d
```

or:

```
n
c\
XXXX
```

Substitute functions

The substitute function(s) changes parts of lines selected by a context search within the line, as in:

(2)s *pattern replacement flags* substitute

The s function replaces part of a line selected by the designated ***pattern*** with the ***replacement*** pattern. The pattern argument contains a ***pattern,*** exactly like the patterns in addresses. The only difference between a pattern and a context address is that a pattern argument may be delimited by any character other than space or newline. By default, only the first string matched by the pattern is replaced, except when you use the **-g** option.

The replacement argument begins immediately after the second delimiting character of the pattern, and must be followed immediately by another instance of the delimiting character. The replacement is not a pattern, and the characters that are special in patterns do not have special meaning in replacement. Instead, the following characters are special:

.	This character is replaced by the string matched by the pattern.
\ *d*	*d* is a single digit that is replaced by the *d*th substring matched by parts of the pattern enclosed in "\(" and "\)". If nested substrings occur in the pattern, the *d*th substring is determined by counting opening delimiters.

As in patterns, you can make special characters literal by preceding them with a backslash (\ ).

A flag argument can contain the following:

g Substitutes the replacement for all non-overlapping instances of the pattern in the line. After a successful substitution, the scan for the next instance of the pattern begins just after the end of the inserted characters; characters put into the line from the replacement are not rescanned.

p Prints the line if a successful replacement was done. The **p** flag causes the line to be written to the output only if a substitution was actually made by the **s** function. Note that if several **s** functions, each followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line are written to the output (one for each successful substitution).

w *file* Writes the line to a file if a successful replacement was done. The **-w** option causes lines tht are actually substituted by the **s** function to be written to the named file. If the filename existed before you run **sed**, it is overwritten; if not, the file is created. A single space must separate **-w** and the filename. The possibilities of multiple, somewhat different copies of one input line being written are the same as for the **-p** option. A combined maximum of ten different filenames can be specified after **w** flags and **w** functions.

Here are some examples. (For the sake of clarity, only the lines affected by the changes are shown. In reality, even unchanged lines are passed through and printed.) When applied to our standard input, the following command:

s/to/by/w changes

produces the following to standard output:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and produces the following in the file *changes*:

```
Through caverns measureless by man
Down by a sunless sea.
```

The following command:

s/[.,;?:]/*P&*/gp

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

With the **g** flag, the following command:

/X/s/an/AN/p

produces:

```
In XANadu did Kubla Khan
```

The following command:

/X/s/an/AN/gp

produces:

```
In XANadu did Kubla KhAN
```

Input-output functions

This section covers the **sed** input-output functions.

p The print function writes the addressed lines to the standard output file at the time **sed** encounters the **p** function, regardless of what succeeding editing commands do to the lines. The maximum number of possible addresses is two.

w The write function writes the addressed lines to *filename.* If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands do to them. Exactly one space must separate the **w** command and the filename. The combined number of write functions and **w** flags can not exceed 10.

r The read function reads the contents of the named file, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line that matched its address. If **r** and **a** functions are executed on the same line, the text from the **a** functions and the **r** functions is written to the output in the order that the functions are executed. Exactly one space must separate the **r** and the filename. One address is possible. If a file mentioned by an **r** function cannot be opened, it is considered a null file rather than an error, and no diagnostic is given.

Note that, since there is a limit to the number of files that can be opened simultaneously, make sure that no more than ten files are mentioned in functions or flags; that number is reduced by one if any **r** functions are present. Only one read file is open at one time.

In the following example, the file *note1* contains the following lines:

```
Note:  Kubla Khan (more properly Kublai Khan;
1216-1294) was the grandson and most eminent
successor of Genghiz (Chingiz) Khan, and
founder of the Mongol dynasty in China.
```

The following command:

/Kubla/r note1

produces:

```
In Xanadu did Kubla Khan
    Note:  Kubla Khan (more properly Kublai Khan;
    1216-1294) was the grandson and most eminent
    successor of Genghiz (Chingiz) Khan, and
    founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Multiple input-line functions

Three upper-case functions deal specially with pattern spaces containing embedded newlines. These functions are intended principally to provide pattern matches across lines in the input.

N	Appends the next input line to the current line in the pattern space; the two input lines are separated by an embedded newline. Pattern matches can extend across the embedded newline(s). There is a maximum of two addresses.
D	Deletes up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), another line is read from the input. In any case, begin the list of editing commands over again. The maximum number of addresses is two.
P	Prints up to and including the first newline in the pattern space. The maximum number of addresses is two.

If there are no embedded newlines in the pattern space, the **P** and **D** functions are equivalent to their lowercase counterparts.

Hold and get functions

These functions save and retrieve part of the input for possible later use:

h The **h** function copies the contents of the pattern space into a holding area, destroying any previous contents of the holding area. The maximum number of addresses is two.

H The **H** function appends the contents of the pattern space to the contents of the holding area. The former and new contents are separated by a newline.

g The **g** function copies the contents of the holding area into the pattern space, destroying the previous contents of the pattern space.

G The **G** function appends the contents of the holding area to the contents of the pattern space. The former and new contents are separated by a newline. The maximum number of addresses is two.

x The exchange command interchanges the contents of the pattern space and the holding area. The maximum number of addresses is two.

For example, apply the following commands to our standard example:

```
1h
1s/ did.*//
1x
G
s/\n/ :/
```

These commands produce the following:

```
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
```

Flow-of-control functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

!	This command causes the next command written on the same line to be applied to only those input lines not selected by the address part. There are two possible addresses.
{	This command causes the next set of commands to be applied or not applied as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping command can appear on the same line as the { or on the next line. The group of commands is terminated by a matching } on a line by itself. Groups can be nested and can have two addresses.
:label	The label function marks a place in the list of editing commands that can be referred to by **b** and **t** functions. The ***label*** can be any sequence of eight or fewer characters; if two different colon functions have identical labels, an error message is generated, and no execution attempted.
b*label*	The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after encountering a colon function with the same label. If no colon function with the same label can be found after all the editing commands have been compiled, an error message is produced, and no execution is attempted. A **b** function with no label is interpreted as a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line. Two addresses are possible.
t*label*	The **t** function tests whether any substitutions have been made successful on the current input line. If so, it branches to the label; if not, it does nothing. The flag that indicates that a successful substitution has been executed is reset either by reading a new input line, or by executing a **t** function.

Miscellaneous functions

There are two other functions of **sed** not discussed in the sections above.

=	The = function writes the number of the line matched by its address to the standard output. One address is possible.
q	The **q** function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated. One address is possible.

Chapter 12
Simple programming with awk

Suppose you want to tabulate some survey results stored in a file, print various reports summarizing these results, generate form letters, reformat a data file for one application package to use with another package, or count the occurrences of a string in a file. Using the **awk**(C) programming language, you can handle these and many other tasks of information retrieval and data processing. The name **awk** is an acronym constructed from the initials of its developers; it denotes the language and also the UNIX system command you use to run an **awk** program.

awk is an easy language to learn. It automatically does quite a few things that you have to program for yourself in other languages. As a result, many useful **awk** programs are only one or two lines long. Because **awk** programs are usually smaller than equivalent programs in other languages, and because they are interpreted, not compiled, **awk** is also a good language for prototyping.

The first part of this chapter introduces you to the basics of **awk** and is intended to make it easy for you to start writing and running your own **awk** programs. The rest of the chapter describes the complete language. For the experienced **awk** user, the end of the chapter contains a summary of the language.

You should be familiar with UNIX operating system commands and shell programming to use this chapter. Although you do not need other programming experience, some knowledge of the C programming language is beneficial because many constructs found in **awk** are also found in C.

Basic awk

This section provides enough information for you to write and run some of your own programs. Each topic presented is discussed in more detail in later sections.

Program structure

The basic operation of **awk** is to scan a set of input lines one after another, searching for lines that match any set of patterns or conditions that you specify. For each pattern, you can specify an action; this action is performed on each line that matches the pattern. Accordingly, an **awk** program is a sequence of pattern-action statements. For example:

Structure:

```
pattern    { action }
pattern    { action }
...
```

Example:

```
$1 == "address"  { print $2, $3 }
```

The example is a typical **awk** program, consisting of one pattern-action statement. The program prints the second and third fields of each input line whose first field is "address." In general, **awk** programs work by matching each line of input against each of the patterns in turn. For each pattern that matches, the associated action (which can involve multiple steps) is executed. Then the next line is read, and the matching starts over. This process typically continues until all the input has been read.

Either the pattern or the action in a pattern-action statement can be omitted. If there is no action with a pattern, the matching line is printed. For example:

```
$1 == "name"
```

If there is no pattern with an action, the action is performed for every input line. For example:

```
{ print $1, $2 }
```

Because patterns and actions are both optional, actions are enclosed in braces to distinguish them from patterns.

Running awk programs

There are two ways to run an **awk** program. First, you can type the command line to execute the pattern-action statements on the set of named input files:

awk ***'pattern-action statements' optional list of input files***

For example, enter:

```
awk '{ print $1, $2 }' file1 file2
```

Notice that the pattern-action statements are enclosed in single quotes. This protects characters like $ from being interpreted by the shell and also allows the program to be longer than one line.

If no files are mentioned on the command line, **awk** reads from the standard input. You can also specify that input comes from the standard input by using the hyphen (-) as one of the input files. For example, to read input first from *file1* and then from the standard input, enter:

```
awk '{ print $3, $4 }' file1 -
```

The arrangement above is convenient when the **awk** program is short. If the program is long, it is often more convenient to put it into a separate file and use the **-f** option to fetch it:

awk -f ***program file   optional list of input files***

For example, the following command line specifies to fetch and execute *myprogram* on input from the file *file1*:

```
awk  -f myprogram file1
```

Fields

Normally, **awk** reads its input one line, or record, at a time; a record is, by default, a sequence of characters ending with a newline character. **awk** then splits each record into fields; by default, a field is a string of non-blank, non-tab characters.

As input for many of the **awk** programs in this chapter, we use the file *countries*, which contains information about the 10 largest countries in the world. Each record contains the name of a country, its area in thousands of square miles, its population in millions, and the continent on which it is found. (Data are from 1978; the U.S.S.R. has been arbitrarily placed in Asia.) The white space between fields is a tab in the original input; a single blank space separates both North and South from America.

```
USSR        8650    262     Asia
Canada      3852    24      North America
China       3692    866     Asia
USA         3615    219     North America
Brazil      3286    116     South America
Australia   2968    14      Australia
India       1269    637     Asia
Argentina   1072    26      South America
Sudan       968     19      Africa
Algeria     920     18      Africa
```

This file is typical of the kind of data **awk** is good at processing — a mixture of words and numbers separated into fields by blanks and tabs.

The number of fields in a record is determined by the field separator. Fields are normally separated by sequences of blanks and/or tabs, so the first record of *countries* has four fields, the second five, and so on. It is possible to set the field separator to just tab, so each line has four fields, matching the meaning of the data. We explain how to do this shortly. For the time being, let's use the default: fields separated by blanks or tabs. The first field within a line is called **$1**, the second **$2**, and so forth. The entire record is called **$0**.

Printing lines

If the pattern in a pattern-action statement is omitted, the action is executed for all input lines. The simplest action is to print each line. You can accomplish this with an **awk** program consisting of a single **print** statement:

```
{ print }
```

The following command line prints each line of *countries*, copying the file to the standard output:

```
awk '{ print }' countries
```

You can also use the **print** statement to print parts of a record. For example, this program prints the first and third fields of each record:

```
{ print $1, $3 }
```

Thus, entering the following command:

```
awk '{ print $1, $3 }' countries
```

produces as output the following sequence of lines:

```
USSR 262
Canada 24
China 866
USA 219
Brazil 116
Australia 14
India 637
Argentina 26
Sudan 19
Algeria 18
```

When printed, items separated by a comma in the **print** statement are separated by the output field separator (by default, a single blank). Each line printed is terminated by the output record separator, (by default, a newline).

NOTE In the remainder of this chapter, we only show **awk** programs, without the command line that invokes them. You can run each complete program, either by enclosing it in quotes as the first argument of the **awk** command, or by putting it in a file and invoking **awk** with the **-f** flag, as discussed in "awk command usage." In an example, if no input is mentioned, the input is assumed to be the file *countries*.

Formatting awk output

For more carefully formatted output, **awk** provides a C-like **printf** statement:

printf *format, expr sub 1, expr sub 2, . . . , expr sub n*

This statement prints the ***expr sub n***'s according to the specification in the string ***format***. For example, the following **awk** program:

{ printf "%10s %6d\n", $1, $3 }

prints the first field ($1) as a string of 10 characters (right justified), then a space, then the third field ($3) as a decimal number in a six-character field, finally a newline (\n). With input from the file *countries*, this program prints an aligned table:

```
      USSR    262
    Canada     24
     China    866
       USA    219
    Brazil    116
 Australia     14
     India    637
 Argentina     26
     Sudan     19
   Algeria     18
```

With **printf,** no output separators or newlines are produced automatically; you must create them yourself by using **\n** in the format specification. "The printf statement" section in this chapter contains a full description of **printf.**

Using simple patterns

You can select specific records for printing or other processing by using simple patterns. **awk** has three kinds of patterns. First, you can use patterns called relational expressions that make comparisons. As an example, the operator == tests for equality. Thus, to print the lines in which the fourth field equals the string Asia, use the program consisting of the following single pattern:

$4 == "Asia"

With the file *countries* as input, this program yields:

```
USSR        8650      262      Asia
China       3692      866      Asia
India       1269      637      Asia
```

The complete set of comparisons are >, >=, <, <=, == (equal to), and != (not equal to). These comparisons can be used to test both numbers and strings. For example, suppose you want to print only countries with a population greater than 100 million. All you need is the following program:

```
$3 > 100
```

(Remember that the third field in the file *countries* is the population in millions.) This program prints all lines in which the third field exceeds 100.

Second, you can use patterns called regular expressions that search for specified characters to select records. The simplest form of a regular expression is a string of characters enclosed in slashes:

```
/US/
```

This program prints each line that contains the (adjacent) letters US anywhere; with the file *countries* as input, it prints:

```
USSR            8650        262         Asia
USA             3615        219         North America
```

There are more examples of regular expressions later in this chapter.

Third, you can use two special patterns, **BEGIN** and **END**, that match before the first record is read and after the last record is processed. This program uses **BEGIN** to print a title:

```
BEGIN  { print "Countries of Asia:" }
/Asia/ { print "     ", $1 }
```

The output from this program is as follows:

```
Countries of Asia:
     USSR
     China
     India
```

Using simple actions

We have already seen the simplest action of an **awk** program: printing each input line. This section explains how you can use built-in and user-defined variables and functions for other simple actions in a program.

Built-in variables

Besides reading the input and splitting it into fields, **awk** counts the number of records read and the number of fields within the current record; you can use these counts in your **awk** programs. The variable **NR** is the number of the current record, and **NF** is the number of fields in the record. For example, the following program prints the number of each line and how many fields it has:

```
{ print NR, NF }
```

This program prints each record preceded by its record number:

```
{ print NR, $0 }
```

User-defined variables

Besides providing built-in variables like **NF** and **NR**, **awk** allows you to define your own variables, which you can use for storing data, doing arithmetic, and the like. To illustrate, consider computing the total population and the average population represented by the data in the file *countries*:

```
    { sum = sum + $3 }
END { print "Total population is", sum, "million"
      print "Average population of", NR, "countries is", sum/NR }
```

NOTE awk initializes *sum* to zero before it is used.

The first action accumulates the population from the third field; the second action, which is executed after the last input, prints the sum and average:

```
Total population is 2201 million
Average population of 10 countries is 220.1
```

Functions

awk has built-in functions that handle common arithmetic and string operations for you. For example, there is an arithmetic function that computes square roots. There is also a string function that substitutes one string for another. **awk** also lets you define your own functions. Functions are described in detail in the section "Actions" in this chapter.

A handful of useful one-liners

Although you can use **awk** used to write large programs of some complexity, many programs are not much more complicated than what we have seen so far. Here is a collection of other short programs that you might find useful and instructive. They are not explained here; however, any new constructs appear later in this chapter.

Print last field of each input line:

```
    { print $NF }
```

Print the tenth input line:

```
    NR == 10
```

Another way of printing only the tenth line is as follows:

```
    sed -n 10p
```

Print the last input line:

```
                   { line = $0}
    END            { print line }
```

Print input lines that do not have four fields:

```
    NF != 4   { print $0, "does not have 4 fields" }
```

Print the input lines with more than four fields:

```
    NF > 4
```

Print the input lines with last field more than 4:

```
    $NF > 4
```

Print the total number of input lines:

```
    END            { print NR }
```

Print total number of fields:

```
                   { nf = nf + NF }
    END            { print nf }
```

Print total number of input characters:

```
                   { nc = nc + length($0) }
    END            { print nc + NR }
```

(Adding **NR** includes in the total the number of newlines.)

Print the total number of lines that contain the string Asia:

```
                   /Asia/       { nlines++ }
                   END          { print nlines }
```

(The statement **nlines++** has the same effect as **nlines = nlines + 1**.)

Error messages

Generally, if you make an error in your **awk** program, you get an error message. For example, if you try to run the following program:

```
    $3 < 200 { print ( $1 }
```

generates the following error messages:

```
awk: syntax error at source line 1
 context is
        $3 < 200 { print ( >>> $1 } <<<
awk: illegal statement at source line 1
         1 extra (
```

Some errors might be detected while your program is running. For example, if you try to divide a number by zero, **awk** stops processing and reports the input record number (**NR**) and the line number in the program.

Patterns

In a pattern-action statement, the pattern is an expression that selects the records for which the associated action is executed. This section describes the kinds of expressions that can be used as patterns.

BEGIN and END

BEGIN and **END** are two special patterns that give you a way to control initialization and wrap-up in an **awk** program. **BEGIN** matches before the first input record is read, so any statements in the action part of a **BEGIN** are done once, before the **awk** command starts to read its first input record. The **END** pattern matches the end of the input, after the last record has been processed.

The following **awk** program uses **BEGIN** to set the field separator to tab (**\t**) and to put column headings on the output. The field separator is stored in a built-in variable called **FS**. Although **FS** can be reset at any time, usually the only sensible place is in a **BEGIN** section, before any input has been read. The program's second **printf** statement, which is executed for each input line, formats the output into a table, neatly aligned under the column headings. The **END** action prints the totals. (Notice that a long line can continue after a comma.)

```
BEGIN { FS = "\t"
    printf "%10s %6s %5s  %s\n",
       "COUNTRY", "AREA", "POP", "CONTINENT" }
   { printf "%10s %6d %5d  %s\n", $1, $2, $3, $4
    area = area + $2; pop = pop + $3 }
END  { printf "\n%10s %6d %5d\n", "TOTAL", area, pop }
```

With the file *countries* as input, this program produces

```
   COUNTRY   AREA   POP  CONTINENT
      USSR   8650   262  Asia
    Canada   3852    24  North America
     China   3692   866  Asia
       USA   3615   219  North America
    Brazil   3286   116  South America
 Australia   2968    14  Australia
     India   1269   637  Asia
 Argentina   1072    26  South America
     Sudan    968    19  Africa
   Algeria    920    18  Africa

     TOTAL  30292  2201
```

Relational expressions

An **awk** pattern can be any expression involving comparisons between strings of characters or numbers. To make comparisons, **awk** includes six relational operators and two regular expression matching operators, ~ (tilde) and !~, (discussed in the next section). Table 12.1 shows these operators and their meanings.

Table 12-1 awk comparison operators

Operator	Meaning
<	less than
<=	less than or equal to
==	equal to
!=	not equal to
>=	greater than or equal to
>	greater than
~	matches
!~	does not match

In a comparison, if both operands are numeric, a numeric comparison is made; otherwise, the operands are compared as strings. (Every value might be either a number or a string; usually **awk** can determine what is intended. The section "Number or string?" contains more information about this.) Thus, the following pattern selects lines where the third field exceeds 100:

```
$3>100
```

This program selects lines that begin with the letters S through Z (i.e. lines with an ASCII value greater than or equal to S):

```
$1 >= "S"
```

The output looks like this:

```
USSR        8650    262     Asia
USA         3615    219     North America
Sudan       968     19      Africa
```

In the absence of any other information, **awk** treats fields as strings. Thus, the following program compares the first and fourth fields as strings of characters:

```
$1 == $4
```

Using the file **countries** as input, this program prints the single line for which this test succeeds:

```
Australia   2968   14   Australia
```

If both fields appear to be numbers, **awk** performs the comparisons numerically.

Regular expressions

awk provides more powerful patterns for searching for strings of characters than the comparisons illustrated in the previous section. These patterns are called *regular expressions* and are like those found in **grep**(C). See the **grep**(C) manual page in the *User's Reference* for more information.

The simplest regular expression is a string of characters enclosed in slashes. For example:

```
/Asia/
```

This program prints all input records that contain the substring Asia; if a record contains Asia as part of a larger string like Asian or Pan-Asiatic it is also printed. In general, if "re" is a regular expression, then the following pattern matches any line that contains a substring specified by the regular expression "re":

```
/re/
```

To restrict a match to a specific field, you use the matching operators ~ (matches) and !~ (does not match). The following program prints the first field of all lines in which the fourth field matches "Asia":

```
$4 ~ /Asia/ { print $1 }
```

This program prints the first field of all lines in which the fourth field does *not* match "Asia":

```
$4 !~ /Asia/ { print $1 }
```

In regular expressions, the following symbols are metacharacters with special meanings like the metacharacters in the UNIX system shell:

```
\ ^ $ . [ ] * + ? ( ) |
```

For example, the metacharacters ^ and $ match the beginning and end of a string, and the metacharacter "dot" (.) matches any single character. Thus, the following matches all records that contain exactly one character:

```
/^.$/
```

A group of characters enclosed in brackets matches any one of the enclosed characters. For example, **/ [ABC] /** matches records containing any one of **A**, **B**, or **C** anywhere. You can abbreviate ranges of letters or digits within brackets. Thus, **/ [a-zA-Z] /** matches any single letter.

If the first character after the left bracket ([) is a caret (^), this complements the class so it matches any character not in the set. Thus, **/ [^a-zA-Z] /** matches any non-letter. The following program prints all records in which the second field is not a string of one or more digits (^ for beginning of string, **[0-9]+** for one or more digits, and **$** for end of string):

```
$2 !~ /^[0-9]+$/
```

Programs of this nature are often used for data validation.

Parentheses ( ) are used for grouping and the pipe symbol (|) is used for alternatives. For example, the following program matches lines containing any one of the four substrings "apple pie," "apple tart," "cherry pie," or "cherry tart:"

```
/(apple | cherry) (pie | tart)/
```

To turn off the special meaning of a metacharacter, precede it by a " \ " (backslash). Thus, the following program prints all lines containing **b** followed by a dollar sign:

```
/b\$/
```

In addition to recognizing metacharacters, the **awk** command recognizes the following C programming language escape sequences within regular expressions and strings:

\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
\ddd	octal value *ddd*
"	quotation mark
\c	any other character *c* literally

For example, to print all lines containing a tab, use the program:

```
/\t/
```

awk interprets any string or variable on the right side of a ~ or !~ as a regular expression. For example, you can write the program:

```
$2 !~ /^[0-9]+$/
```

as:

```
BEGIN   { digits = "^[0-9]+ $" }
$2 !~ digits
```

Suppose you wanted to search for a string of characters such as **^[0-9]+$**. When a literal quoted string like **"^[0-9]+$"** is used as a regular expression, one extra level of backslashes is needed to protect regular expression metacharacters. This is because one level of backslashes is removed when a string is originally parsed. If a backslash is needed in front of a character to turn off its special meaning in a regular expression, then that backslash needs a preceding backslash to protect it in a string.

For example, suppose we want to match strings containing **b** followed by a dollar sign. The regular expression for this pattern is **b\$**. To create a string to represent this regular expression, add one more backslash: **"b\\$"**. The two regular expressions on each of the following lines are equivalent:

```
x~"b\\$"        x~/b\$/
x~"b\$"         x~/b$/
x~"b$"          x~/b$/
x~"\\t"         x~/\t/
```

The precise form of regular expressions and the substrings they match is given in Table 12.2. The unary operators *, +,, and ? have the highest precedence, with concatenation next, and then alternation |. All operators are left associative. The *r* stands for any regular expression.

Table 12-2 awk regular expressions

Expression	Matches
c	any non-metacharacter *c*
\c	character *c* literally
^	beginning of string
$	end of string
.	any character but newline
[*s*]	any character in set *s*
[^*s*]	any character not in set *s*
*r**	zero or more *r*'s
r+	one or more *r*'s
r?	zero or one *r*
(*r*)	*r*
r sub* 1 *r sub 2	***r sub* 1** then ***r sub*** 2 (concatenation)
r sub* 1** | ***r sub 2	***r sub* 1** or ***r sub*** 2 (alternation)

Combining patterns

A compound pattern combines simpler patterns with parentheses and the logical operators || (or), && (and), and ! (not). For example, if you want to print all countries in Asia with a population of more than 500 million, use the following program. This program selects all lines in which the fourth field is Asia and the third field exceeds 500:

```
$4 == "Asia" && $3 > 500
```

The following program selects lines with Asia or Africa as the fourth field:

```
$4 == "Asia" || $4 == "Africa"
```

Another way to write the latter query is to use a regular expression with the alternation operator **|**:

$4 ~ /^(Asia | Africa)$/

The negation operator **!** has the highest precedence, then **&&,**, and finally **| |**. The operators **&&** and **| |** evaluate their operands from left to right; evaluation stops as soon as truth or falsehood is determined.

Pattern ranges

A pattern range consists of two patterns separated by a comma:

pat sub 1, pat sub 2   { . . . }

In this case, the action is performed for each line between an occurrence of ***pat sub 1*** and the next occurrence of ***pat sub*** 2 (inclusive). As an example, the following pattern:

/Canada/, /Brazil/

matches lines starting with the first line that contains the string Canada, up through the next occurrence of the string Brazil:

```
Canada          3852        24          North America
China           3692        866         Asia
USA             3615        219         North America
Brazil          3286        116         South America
```

Similarly, because **FNR** is the number of the current record in the current input file (and *FILENAME* is the name of the current input file), the following program prints the first five records of each input file with the name of the current input file prepended:

FNR == 1, FNR == 5 { print FILENAME, $0 }

Actions

In a pattern-action statement, the action determines what is to be done with the input records that the pattern selects. Actions frequently are simple printing or assignment statements, but they can also be a combination of one or more statements. This section describes the statements that can make up actions.

Built-in awk variables

Table 12.3 lists the built-in variables that **awk** maintains. You are familiar with some of these; others are used in this and later sections.

Table 12-3 awk built-in variables

Variable	Meaning	Default
ARGC	number of command-line arguments	-
ARGV	array of command-line arguments	-
FILENAME	name of current input file	-
FNR	record number in current file	-
FS	input field separator	blank&tab
NF	number of fields in current record	-
NR	number of records read so far	-
OFMT	output format for numbers	%.6g
OFS	output field separator	blank
ORS	output record separator	newline
RS	input record separator	newline
RSTART	index of first character matched by **match()**	-
RLENGTH	length of string matched by **match()**	-
SUBSEP	subscript separator	"\034"

Performing arithmetic

Actions can use conventional arithmetic expressions to compute numeric values. As a simple example, suppose we want to print the population density for each country in the file *countries*. Because the second field is the area in thousands of square miles, and the third field is the population in millions, the expression **1000 * $3 / $2** gives the population density in people per square mile. Use the following program to print the name of each country and its population density:

```
    { printf "%10s %6.1f\n", $1, 1000 * $3 / $2 }
```

The output looks like this:

```
        USSR   30.3
      Canada    6.2
       China  234.6
         USA   60.6
      Brazil   35.3
   Australia    4.7
       India  502.0
   Argentina   24.3
       Sudan   19.6
     Algeria   19.6
```

Arithmetic is done internally in floating point. The arithmetic operators are +, -, *, /, % (remainder), and ^ (exponentiation; ** is a synonym). Arithmetic expressions can be created by applying these operators to constants, variables, field names, array elements, functions, and other expressions, all of which are discussed later. Note that **awk** recognizes and produces scientific (exponential) notation: 1e6, 1E6, 10e5, and 1000000 are numerically equal.

awk has assignment statements like those found in the C programming language. The simplest form is the assignment statement:

v = *e*

where *v* is a variable or field name, and *e* is an expression. For example, to compute the number of Asian countries and their total populations, use this program:

```
$4 == "Asia"  { pop = pop + $3; n = n + 1 }
END           { print "population of", n,
                 "Asian countries in millions is", pop }
```

Applied to *countries*, this program produces:

```
population of 3 Asian countries in millions is 1765
```

The action associated with the pattern $4 == "Asia" contains two assignment statements, one to accumulate population and the other to count countries. The variables are not explicitly initialized, yet everything works properly because **awk** initializes each variable with the string value "" and the numeric value 0.

The assignments in the previous program can be written more concisely using the operators += and ++ as follows:

```
$4 == "Asia"        { pop += $3; ++n }
```

The += operator is borrowed from the C programming language:

```
pop += $3
```

It has the same effect as:

```
pop = pop + $3
```

The += operator is shorter and runs faster. The same is true of the ++ operator, which adds one to a variable.

The abbreviated assignment operators are +=, -=, *=, /=, %=, and ^=. Their meanings are similar. For example,

v op= e

has the same effect as

v = v op e

The increment operators are ++ and -. As in C, you can use them as prefix (++x) or postfix (x++) operators. If x is 1, then **i=++x** increments **x**, then sets **i** to 2. On the other hand, **i=x++** sets **i** to 1, then increments **x**. An analogous interpretation applies to prefix - and postfix -.

Assignment and increment and decrement operators can all be used in arithmetic expressions.

We use default initialization to advantage in the following program, which finds the country with the largest population:

```
maxpop < $3  { maxpop = $3; country = $1 }
END          { print country, maxpop }
```

Note, that this program is not correct if all values of **$3** are negative.

awk provides the built-in arithmetic functions shown in Table 12.4.

Table 12-4 awk built-in arithmetic functions

Function	Value Returned
atan2(*y*,*x*)	arctangent of *y*/^*x* in the range - *pi* to *pi*
cos(*x*)	cosine of *x*, with *x* in radians
exp(*x*)	exponential function of *x*
int(*x*)	integer part of *x* truncated towards 0
log(*x*)	natural logarithm of *x*
rand()	random number between 0 and 1
sin(*x*)	sine of *x*, with *x* in radians
sqrt(*x*)	square root of *x*
srand(*x*)	*x* is new seed for rand()

Both *x* and *y* are arbitrary expressions. The function **rand()** returns a pseudo-random floating point number in the range (0,1), and **srand(*x*)** can be used to set the seed of the generator. If **srand()** has no argument, the seed is derived from the time of day.

Using strings and string functions

A string constant is created by enclosing a sequence of characters inside quotation marks, as in **"abc"** or **"hello, everyone"**. String constants can contain the C programming language escape sequences for special characters listed in "Regular expressions" in this chapter.

String expressions are created by concatenating constants, variables, field names, array elements, functions, and other expressions. The following program prints each record preceded by its record number and a colon, with no blanks:

```
{ print NR ":" $0 }
```

This concatenates the three strings representing the record number, the colon, and the record, and prints the resulting string. The concatenation operator has no explicit representation other than juxtaposition.

awk provides the built-in string functions shown in Table 12.5. In this table, "r" represents a regular expression (either as a string or as */r/*), *s*, and *t* are string expressions, and *n* and *p* are integers.

Table 12-5 awk built-in string functions

Function	Description
gsub(*r, s*)	substitutes ***s*** for ***r*** globally in current record, returns number of substitutions
gsub(*r, s, t*)	substitutes ***s*** for ***r*** globally in string ***t***, returns number of substitutions
index(*s, t*)	returns position of string ***t*** in ***s***, 0 if not present
length(*s*)	returns length of ***s***
match(*s, r*)	returns the position in ***s*** where ***r*** occurs, 0 if not present
split(*s, a*)	splits ***s*** into array ***a*** on **FS**, returns number of fields
split(*s, a, r*)	splits ***s*** into array ***a*** on ***r***, returns number of fields
sprintf(*fmt, expr-list*)	returns ***expr-list*** formatted according to format string ***fmt***
sub(*r, s*)	substitutes ***s*** for first ***r*** in current record, returns number of substitutions
sub(*r, s, t*)	substitutes ***s*** for first ***r*** in ***t***, returns number of substitutions
substr(*s, p*)	returns suffix of ***s*** starting at position ***p***
substr(*s, p, n*)	returns substring of ***s*** of length ***n*** starting at position ***p***

The functions **sub** and **gsub** are patterned after the substitute command in the text editor **ed**(C). See the *User's Reference* for more information. The function **gsub(*r, s, t*)** replaces successive occurrences of substrings matched by the regular expression ***r*** with the replacement string *s* in the target string ***t***. (As in **ed**, the left-most match is used and is made as long as possible.) **gsub** returns the number of substitutions made. The function **gsub(*r, s*)** is a synonym for **gsub(*r, s*, $0)**. For example, the following program transcribes its input, replacing occurrences of USA with United States:

```
{ gsub(/USA/, "United States"); print }
```

The **sub** functions are similar, except that they only replace the first matching substring in the target string.

The function **index(*s, t*)** returns the left-most position where the string ***t*** begins in *s*, or zero if ***t*** does not occur in *s*. The first character in a string is at position 1. For example, the following command returns 2:

```
index("banana", "an")
```

The **length** function returns the number of characters in its argument string; thus, the following prints each record, preceded by its length:

```
{ print length($0), $0 }
```

(**$0** does not include the input record separator.) The following program prints the longest country name (**Australia**):

```
length($1) > max  { max = length($1); name = $1 }
END               { print name }
```

The **match(*s, r*)** function returns the position in string *s* where regular expression ***r*** occurs, or 0 if it does not occur. This function also sets two built-in variables **RSTART** and **RLENGTH**. **RSTART** is set to the starting position of the match in the string; this is the same value as the returned value. **RLENGTH** is set to the length of the matched string. (If a match does not occur, **RSTART** is 0, and **RLENGTH** is -1.) For example, the following program finds the first occurrence of the letter "i," followed by at most one character, followed by the letter "a" in a record:

```
{ if (match($0, /i.?a/))
    print RSTART, RLENGTH, $0 }
```

This program produces the following output on the file *countries*:

```
217  2  USSR       8650  262  Asia
26   3  Canada     3852  24   North America
3    3  China      3692  866  Asia
24   3  USA        3615  219  North America
27   3  Brazil     3286  116  South America
8    2  Australia  2968  14   Australia
4    2  India      1269  637  Asia
7    3  Argentina  1072  26   South America
17   3  Sudan      968   19   Africa
6    2  Algeria    920   18   Africa
```

NOTE **match( )** matches the left-most longest matching string. For example, if you use the following record as input:

    AsiaaaAsiaaaaan

this program:

```
{ if (match($0, /a+/)) print RSTART, RLENGTH, $0 }
```

matches the first string of **a**'s and sets **RSTART** to 4 and **RLENGTH** to 3.

The following function:

sprintf(*format, expr sub 1, expr sub 2,. . . ,*

returns (without printing) a string containing:

expr sub 1, expr sub 2, . . . , expr sub n

formatted according to the **printf** specifications in the string *format*. "The printf statement" in this chapter contains a complete specification of the format conventions.

The statement:

```
x = sprintf("%10s %6d", $1, $2)
```

assigns to x the string produced by formatting the values of **$1** and **$2** as a 10-character string and a decimal number in a field of width at least six; **x** can be used in any subsequent computation.

The function **substr**(*s, p, n*) returns the substring of *s* that begins at position *p* and is at most *n* characters long. If **substr**(*s, p*) is used, the substring goes to the end of *s*; that is, it consists of the suffix of *s* beginning at position *p*. For example, we could abbreviate the country names in *countries* to their first three characters by invoking the following program:

```
{ $1 = substr($1, 1, 3); print }
```

This produces the following output:

```
USS 8650 262 Asia
Can 3852 24 North America
Chi 3692 866 Asia
USA 3615 219 North America
Bra 3286 116 South America
Aus 2968 14 Australia
Ind 1269 637 Asia
Arg 1072 26 South America
Sud 968 19 Africa
Alg 920 18 Africa
```

Note that setting **$1** in the program forces **awk** to recompute **$0** and, therefore, the fields are separated by blanks (the default value of **OFS**), not by tabs.

Strings are stuck together (concatenated) merely by writing them one after another in an expression. For example, when invoked on file *countries*, the following program:

```
    { s = s substr($1, 1, 3) " " }
END { print s }
```

prints the following by building s up, one piece at a time, from an initially empty string:

```
USS Can Chi USA Bra Aus Ind Arg Sud Alg
```

Field variables

The fields of the current record can be referred to by the field variables **$1, $2, . . . , $NF**. Field variables share all of the properties of other variables: they can be used in arithmetic or string operations, and they can have values assigned to them. So, for example, you can divide the second field of the file *countries* by 1000 to convert the area from thousands to millions of square miles:

```
{ $2 /= 1000; print }
```

You can also assign a new string to a field:

```
BEGIN              { FS = OFS = "\t" }
$4 == "North America"  { $4 = "NA" }
$4 == "South America"  { $4 = "SA" }
                   { print }
```

The **BEGIN** action in this program resets the input field separator **FS** and the output field separator **OFS** to a tab. Notice that the *print* in the fourth line of the program prints the value of **$0** after it has been modified by previous assignments.

Fields can be accessed by expressions. For example, **$(NF-1)** is the second to last field of the current record. The parentheses are needed to show that the value of **$NF-1** is 1 less than the value in the last field.

A field variable referring to a nonexistent field, for example, **$(NF+1)**, has as its initial value the empty string. A new field can be created, however, by assigning a value to it. For example, the following program invoked on the file *countries* creates a fifth field giving the population density:

```
BEGIN { FS = OFS = "\t" }
    { $5 = 1000 * $3 / $2; print }
```

The number of fields can vary from record to record, but there is usually an implementation limit of 100 fields per record.

Number or string?

Variables, fields, and expressions can have a numeric value, a string value, or both at any time. They take on numeric or string values according to context. For example, in the context of an arithmetic expression like the following:

```
pop += $3
```

pop and **$3** must be treated numerically, so their values can be coerced to numeric type if necessary.

In a string context like:

```
print $1 ":" $2
```

$1 and **$2** must be strings to be concatenated, so they can be coerced if necessary.

In an assignment *v=e* or *v ~op=e*, the type of *v* becomes the type of ***e***. In an ambiguous context like the following, the type of the comparison depends on whether the fields are numeric or string:

```
$1 == $2
```

This can only be determined when the program runs; it might differ from record to record.

In comparisons, if both operands are numeric, the comparison is numeric; otherwise, operands are coerced to strings, and the comparison is made on the string values. All field variables are of type string; in addition, each field that contains only a number is also considered numeric. This determination is done at run time. For example, the comparison "$1 == $2" succeeds on any pair of the following inputs:

```
1  1.0  +1  0.1e+1  10E-1  001
```

but fails on these inputs:

```
(null)          0
(null)          0.0
0a              0
1e50            1.0e50
```

There are two idioms for coercing an expression of one type to the other:

number ""	concatenate a null string to a ***number*** to coerce it to type string
string + 0	add zero to a ***string*** to coerce it to type numeric

Thus, to force a string comparison between two fields, such as:

```
$1 "" == $2 ""
```

The numeric value of a string is the value of any prefix of the string that looks numeric; thus the value of **12.34x** is 12.34, while the value of **x12.34** is zero. The string value of an arithmetic expression is computed by formatting the string with the output format conversion **OFMT**.

Uninitialized variables have numeric value 0 and string value "". Nonexistent fields and fields that are explicitly null have only the string value ""; they are not numeric.

Control flow statements

awk provides **if-else**, **while**, **do-while**, and **for** statements, and statement grouping with braces, as in the C programming language.

The **if** statement syntax is

> **if** (***expression***) ***statement sub 1*** else ***statement sub 2***

The ***expression*** acting as the conditional has no restrictions; it can include the relational operators **<**, **<=**, **>**, **>=**, **==**, and **!=**; the regular expression matching operators **~** and **!~** ; the logical operators **||**, **&&**, and **!**; juxtaposition for concatenation; and parentheses for grouping.

In the **if** statement, the ***expression*** is first evaluated. If it is non-zero and non-null, ***statement sub 1*** is executed; otherwise ***statement sub 2*** is executed. The **else** part is optional.

A single statement can always be replaced by a statement list enclosed in braces. The statements in the statement list are terminated by newlines or semicolons.

Rewriting the maximum population program from "Arithmetic functions" with an **if** statement results in the following:

```
{  if (maxpop < $3) {
      maxpop = $3
      country = $1
   }
}
END { print country, maxpop }
```

The **while** statement is exactly that of the C programming language:

while (*expression*) *statement*

The ***expression*** is evaluated; if it is non-zero and non-null, the ***statement*** is executed, and the ***expression*** is tested again. The cycle repeats as long as the ***expression*** is non-zero. For example, use the following to print all input fields one per line:

```
{ i = 1
  while (i <= NF) {
    print $i
    i++
  }
}
```

The **for** statement is like that of the C programming language:

for (*expression sub 1* ; *expression* ; *expression sub 2*) *statement*

This has the same effect as:

```
expression sub 1
while (expression) {
                  statement
                  expression sub 2
}
```

Thus, the following statement:

```
{ for (i = 1; i <= NF; i++)  print $i }
```

does the same job as the **while** example above. An alternate version of the **for** statement is described in the next section.

The **do** statement has the following form:

do *statement* while (*expression*)

The ***statement*** is executed repeatedly until the value of the ***expression*** becomes zero. Because the test takes place after the execution of the ***statement*** (at the bottom of the loop), it is always executed at least once. As a result, the **do** statement is used much less often than **while** or **for**, which test for completion at the top of the loop.

The following example of a **do** statement prints all lines except those between *start* and *stop*:

```
/start/ {
      do {
                  getline x
        } while (x !~ /stop/)
     }
     { print }
```

The **break** statement causes an immediate exit from an enclosing **while** or **for**; the **continue** statement causes the next iteration to begin. The **next** statement causes **awk** to skip immediately to the next record and begin matching patterns starting from the first pattern-action statement.

The **exit** statement causes the program to behave as if the end of the input had occurred; no more input is read, and the END action, if any, is executed. Within the END action, the following statement causes the program to return the value of ***expr*** as its exit status:

```
exit expr
```

If there is no ***expr***, the exit status is zero.

Arrays

awk provides one-dimensional arrays. You do not need to declare arrays and array elements; like variables, they spring into existence when you use them. An array subscript can be a number or a string.

As an example of a conventional numeric subscript, the following statement assigns the current input line to the NRth element of the array x:

```
x[NR] = $0
```

In fact, it is possible in principle (though perhaps slow) to read the entire input into an array with the **awk** program like this:

```
 { x[NR] = $0 }
END { . . . processing . . . }
```

The first action merely records each input line in the array x, indexed by line number; processing is done in the END statement.

Array elements can also be named by nonnumeric values. For example, the following program accumulates the total population of Asia and Africa into the associative array **pop**. The END action prints the total population of these two continents.

```
/Asia/      { pop["Asia"] += $3 }
/Africa/    { pop["Africa"] += $3 }
END         { print "Asian population in millions is", pop["Asia"]
              print "African population in millions is",
            pop["Africa"] }
```

On the file *countries,* this program generates the following output:

```
Asian population in millions is 1765
African population in millions is 37
```

In this program, if we use **pop[Asia]** instead of **pop["Asia"]**, the expression uses the value of the variable Asia as the subscript. because the variable is uninitialized, the values would have been accumulated in **pop[""]**.

Suppose our task is to determine the total area in each continent of the file *countries*. Any expression can be used as a subscript in an array reference. Thus, this statement:

```
area[$4] += $2
```

uses the string in the fourth field of the current input record to index the array **area** and in that entry accumulates the value of the second field:

```
BEGIN       { FS = "\t" }
            { area[$4] += $2 }
END         { for (name in area)
                  print name, area[name] }
```

When you invoke this on the *countries* file, this program produces the following output:

```
Africa 1888
North America 7467
South America 4358
Asia 13611
Australia 2968
```

This program uses a form of the **for** statement that iterates over all defined subscripts of an array:

for (*i* in *array*) *statement*

This executes ***statement*** with the variable ***i*** set in turn to each value of ***i*** for which *array[i]* has been defined. The loop is executed once for each defined subscript, which are chosen in a random order. Results are unpredictable when ***i*** or ***array*** is altered during the loop.

awk does not provide multi-dimensional arrays, but it does permit a list of subscripts. They are combined into a single subscript with the values separated by an unlikely string (stored in the variable **SUBSEP**). For example, the following code creates an array that behaves like a two-dimensional array; the subscript is the concatenation of **i, SUBSEP,** and **j**:

```
for (i = 1; i <= 10; i++)
            for (j = 1; j <= 10; j++)
               arr[i,j] = ...
```

You can determine whether a particular subscript ***i*** occurs in an array **arr** by testing the condition ***i*** in **arr**:

```
if ("Africa" in area) ...
```

This condition performs the test without the side effect of creating **area["Africa"]**, which would happen if we used the following:

```
if (area["Africa"] != "") ...
```

Note that neither is a test of whether the array **area** contains an element with value **"Africa"**.

It is also possible to split any string into fields in the elements of an array using the built-in function **split**. The function splits the string **s1:s2:s3** into three fields (using the separator :) and stores **s1** in **a[1]**, **s2** in **a[2]**, and **s3** in **a[3]**.

```
split("s1:s2:s3", a, ":")
```

The number of fields found, here three, is returned as the value of **split**. The third argument of **split** is a regular expression to be used as the field separator. If the third argument is missing, **FS** is used as the field separator.

An array element can be deleted with the **delete** statement:

delete ***arrayname*****[*****subscript*****]**

User-defined functions

awk provides user-defined functions. A function is defined as:

```
function name(argument-list){
            statements
}
```

The definition can occur anywhere a pattern-action statement can. The argument list is a list of variable names separated by commas; within the body of the function, these variables refer to the actual parameters when the function is called. There must be no space between the function name and the left parenthesis of the argument list when the function is called; otherwise it looks like a concatenation. For example, the following program defines and tests the usual recursive factorial function (using some input other than the file *countries*):

```
function fact(n) {
  if (n <= 1)
    return 1
  else
    return n * fact(n-1)
}
{ print $1 "! is " fact($1) }
```

Array arguments are passed by reference, as in C, so it is possible for the function to alter array elements or create new ones. Scalar arguments are passed by value, however, so the function cannot affect their values outside. Within a function, formal parameters are local variables, but all other variables are global. (You can have any number of extra formal parameters that are used purely as local variables.) The **return** statement is optional, but the returned value is undefined if it is not included.

Some lexical conventions

Comments may be placed in **awk** programs; they begin with the character "#" and end at the end of the line. For example:

```
print x, y    # this is a comment
```

Statements in an **awk** program normally occupy a single line. Several statements can occur on a single line if they are separated by semicolons. You can continue a long statement over several lines by terminating each continued line by a backslash. (It is not possible to continue a "..." string.) This explicit continuation is rarely necessary, however, because statements continue automatically after the operators **&&** and **| |** or if the line ends with a comma (for example, as might occur in a **print** or **printf** statement).

Several pattern-action statements can appear on a single line if separated by semicolons.

awk output

The **print** and **printf** statements are the two primary constructs that generate output. The **print** statement is used to generate simple output; **printf** is used for more carefully formatted output. Like the shell, **awk** lets you redirect output so that output from **print** and **printf** can be directed to files and pipes. This section describes how to use these two statements.

The print statement

The following statement prints the string value of each expression separated by the output field separator followed by the output record separator:

print ***expr sub 1, expr sub 2, . . . , expr sub n***

The following statement:

```
print
```

is an abbreviation for:

```
print $0
```

To print an empty line, use the following:

```
print ""
```

Output separators

The built-in variables **OFS** and **ORS** contain the output field separator and record separator. Initially, **OFS** is set to a single blank and **ORS** to a single newline, but you can change these values at any time. For example, the following program prints the first and second fields of each record with a colon between the fields and two newlines after the second field:

```
BEGIN { OFS = ":"; ORS = "\n\n" }
    { print $1, $2 }
```

Notice that the following program prints the first and second fields with no intervening output field separator:

```
{ print $1 $2 }
```

This is because **$1 $2** is a string consisting of the concatenation of the first two fields.

The printf statement

awk's **printf** statement is the same as that in C except that the * format specifier is not supported. The **printf** statement has the general form:

printf ***format, expr sub 1, expr sub 2, . . . , expr sub n***

where ***format*** is a string that contains both information to be printed and specifications on what conversions to perform on the expressions in the argument list, as in Table 12.6. Each specification begins with a "%", ends with a letter that determines the conversion, and can include:

-	left-justify expression in its field
width	pad field to this width as needed; fields that begin with a leading 0 are padded with zeros
.prec	maximum string width or digits to right of decimal point

Table 12-6 awk printf conversion characters

Character	Prints expression as
%c	single character
%d	decimal number
%e	**[-]d.ddddddE[+-]dd**
%f	**[-]ddd.dddddd**
%g	**e** or **f** conversion, whichever is shorter, with nonsignificant zeros suppressed
%o	unsigned octal number
%s	string
%x	unsigned hexadecimal number
%%	print a %; no argument is converted

Here are some examples of **printf** statements with the corresponding output:

printf "%d", 99/2	`49`
printf "%e", 99/2	`4.950000e+01`
printf "%f", 99/2	`49.500000`
printf "%6.2f", 99/2	`49.50`
printf "%g", 99/2	`49.5`
printf "%o", 99	`143`
printf "%06o", 99	`000143`
printf "%x", 99	`63`
printf "|%s|", "January"	`|January|`
printf "|%10s|", "January"	`|   January|`
printf "|%-10s|", "January"	`|January   |`
printf "|%.3s|", "January"	`|Jan|`
printf "|%10.3s|", "January"	`|       Jan|`
printf "|%-10.3s|", "January"	`|Jan       |`
printf "%%"	`%`

The default output format of numbers is **%.6g**; this can be changed by assigning a new value to **OFMT**. **OFMT** also controls the conversion of numeric values to strings for concatenation and creation of array subscripts.

Output into files

You can print output into files, instead of to the standard output using the > and >> redirection operators. For example, if you invoke the following program on the file *countries,* **awk** prints all lines where the population (third field) is bigger than 100 into a file called *bigpop,* and all other lines into *smallpop*:

```
$3 > 100  { print $1, $3 >"bigpop" }
$3 <= 100 { print $1, $3 >"smallpop" }
```

Notice that the filenames must be quoted; without quotes, *bigpop* and *smallpop* are merely uninitialized variables. If the output filenames are created by an expression, they also must be enclosed in parentheses:

```
$4 ~ /North America/ { print $1 > ("tmp" FILENAME) }
```

This is because the > operator has higher precedence than concatenation; without parentheses, the concatenation of *tmp* and *FILENAME* does not work.

NOTE Files are opened once in an **awk** program. If > is used to open a file, its original contents are overwritten. But if >> is used to open a file, its contents are preserved and the output is appended to the file. Once the file has been opened, the two operators have the same effect.

Output into pipes

You can also direct printing into a pipe with a command on the other end, instead of into a file. The following statement causes the output of **print** to be piped into the ***command-line***:

```
print | "command-line"
```

Although we show the ***command-line*** and filenames here as literal strings enclosed in quotes, they can also come from variables, and the return values from functions.

Suppose we want to create a list of continent-population pairs, and sort it alphabetically by continent. The **awk** program below accumulates the population values in the third field for each of the distinct continent names in the fourth field in an array called **pop**. Then it prints each continent and its population, and pipes this output into the **sort** command:

```
BEGIN  { FS = "\t" }
       { pop[$4] += $3 }
END    { for (c in pop)
            print c ":" pop[c] | "sort" }
```

Invoked on the file *countries*, this program yields the following:

```
Africa:37
Asia:1765
Australia:14
North America:243
South America:142
```

In all of these **print** statements involving redirection of output, the files or pipes are identified by their names (that is, the pipe above is literally named *sort*), but they are created and opened only once in the entire run. So, in the last example, for all **c** in **pop**, only one sort pipe is open.

There is a limit to the number of files that can be open simultaneously. The statement **close(*file*)** closes a file or pipe; ***file*** is the string used to create it in the first place. For example:

```
close("sort")
```

When opening or closing a file, different strings are different commands.

Input

The most common way to give input to an **awk** program is to name on the command line the file(s) that contains the input. This is the method that we have been using in this chapter. However, you can use several other methods, each of which is described in this section.

Files and pipes

You can provide input to an **awk** program by putting the input data into a file, say *awkdata*, and then executing it like this:

awk ***'program' awkdata***

awk reads its standard input if no filenames are given (see "Usage" in this chapter); thus, a second common arrangement is to have another program pipe its output into **awk**. For example, **grep**(C) selects input lines containing a specified regular expression, but it can do so faster than **awk**, because this is the only thing it does. Therefore, you can invoke the pipe like this:

```
grep 'Asia' countries | awk '...'
```

grep quickly finds the lines containing Asia and passes them on to the **awk** program for subsequent processing.

Input separators

With the default setting of the field separator **FS**, input fields are separated by blanks or tabs, and leading blanks are discarded, so each of these lines has the same first field:

```
        field1   field2
      field1
    field1
```

When the field separator is a tab, however, leading blanks are not discarded.

The field separator can be set to any regular expression by assigning a value to the built-in variable **FS**. For example the following sets it to an optional comma followed by any number of blanks and tabs:

BEGIN { FS = "(,[\\t]*) | ([\\t]+)" }

You can also set **FS** on the command line with the **-F** argument. For example, this behaves the same as the previous example:

awk -F'(,[\t]*) | ([\t]+)' '...'

Regular expressions used as field separators match the left-most longest occurrences (as in **sub()**), but they do not match null strings.

Multi-line records

Records are normally separated by newlines, so that each line is a record; you can change this, but only in a limited way. If you set the built-in record separator variable **RS** to an empty string like this:

BEGIN { RS = "" }

In this case, input records can be several lines long; a sequence of empty lines separates records. A common way to process multiple-line records is to set the record separator to an empty line and the field separator to a newline, as in the following example:

BEGIN { RS = ""; FS = "\n" }

There is a limit to how long a record can be; it is usually about 2500 characters. "The getline function" and "Co-operation with the shell" in this chapter show other examples of processing multi-line records.

The getline function

For some tasks, **awk**'s facility for automatically breaking its input into records that are more than one line long is not adequate for some tasks. For example, if records are not separated by blank lines, but by something more complicated, merely setting **RS** to null does not work. In such cases, it is necessary to manage the splitting of each record into fields in the program. Here are some suggestions.

Use the function **getline** to read input either from the current input or from a file or pipe, by using redirection in a manner analogous to **printf**. By itself, **getline** fetches the next input record and performs the normal field-splitting operations on it. It sets **NF**, **NR**, and **FNR**. **getline** returns 1 if there was a record present, 0 if the end-of-file was encountered, and -1 if some error occurred (such as failure to open a file).

To illustrate, suppose we have input data consisting of multi-line records, each of which begins with a line beginning with **START** and ends with a line beginning with **STOP**. The following **awk** program processes these multi-line records, a line at a time, putting the lines of the record into consecutive entries of an array:

```
f[1] f[2] ... f[nf]
```

Once the line containing **STOP** is encountered, the record can be processed from the data in the **f** array:

```
/^START/ {
      f[nf=1] = $0
      while (getline && $0 !~ /^STOP/)
         f[++nf] = $0
      # now process the data in f[1] ... f[nf]
      ...
}
```

Notice that this code uses the fact that **&&** evaluates its operands left to right and stops as soon as one is true. The same job can also be done by the following program:

```
/^START/ && nf==0     { f[nf=1] = $0 }
nf > 1                { f[++nf] = $0 }
/^STOP/               { # now process the data in f[1] ... f[nf]
                        ...
                        nf = 0
}
```

The following statement reads the next record into the variable **x**:

```
getline x
```

No splitting is done; **NF** is not set.

This statement reads from *file* instead of the current input:

```
getline <"file"
```

It has no effect on **NR** or **FNR**, but field splitting is performed, and **NF** is set.

The following statement gets the next record from *file* into **x**:

```
getline x <"file"
```

In this case, no splitting is done, and **NF**, **NR**, and **FNR** are untouched.

NOTE If a filename is an expression, it should be in parentheses for evaluation:

```
while ( getline x < (ARGV[1] ARGV[2]) ) { ...}
```

This is because the < has precedence over concatenation. Without parentheses, a statement such as the following sets *x* to read the file *tmp <value of FILENAME*:

```
getline x < "tmp" FILENAME
```

Also, if you use this **getline** statement form, a statement like the following loops forever if the file cannot be read:

```
while ( getline x < file ) {...}
```

This is because **getline** returns -1, not zero, if an error occurs. A better way to write this test is as follows:

```
while ( getline x < file > 0) {...}
```

It is also possible to pipe the output of another command directly into **getline.** For example, the following statement executes **who** and pipes its output into **getline:**

```
while ("who" | getline)
  n++
```

Each iteration of the **while** loop reads one more line and increments the variable n, so after the **while** loop terminates, **n** contains a count of the number of users. Similarly, the following statement pipes the output of **date** into the variable **d,** thus setting **d** to the current date:

```
"date" | getline d
```

Table 12.7 summarizes the **getline** function.

Table 12-7 getline function

Form	Sets
getline	$0, NF, NR, FNR
getline *var*	*var,* NR, FNR
getline <*file*	$0, NF
getline *var* <*file*	*var*
cmd | getline	$0, NF
cmd | getline *var*	*var*

Command-line arguments

The command-line arguments are available to an **awk** program: the array **ARGV** contains the elements **ARGV[0]**, . . . , **ARGV[ARGC-1]**; as in C, **ARGC** is the count. **ARGV[0]** is the name of the program (generally **awk**); the remaining arguments are whatever was provided (excluding the program and any optional arguments).

The following command line contains an **awk** program that echoes the arguments that appear after the program name:

```
awk '
BEGIN {
  for (i = 1; i < ARGC; i++)
      printf "%s ", ARGV[i]
  printf "\n"
}' $*
```

You can modify or add to the arguments; you can also alter **ARGC**. As each input file ends, **awk** treats the next non-null element of **ARGV** (up to the current value of **ARGC-1**) as the name of the next input file.

The one exception to the rule that an argument is a filename is that if it is of the following form, then the variable ***var*** is set to the value ***value*** as if by assignment:

var=value

If ***value*** is a string, no quotes are needed. Such an argument is not treated as a filename.

Using awk with other commands and the shell

awk gains its greatest power when you use it in conjunction with other programs. Here we describe some of the ways in which **awk** programs co-operate with other commands.

The system function

The built-in function **system** (***command-line***) executes the command ***command-line,*** which can be a string computed by, for example, the built-in function **sprintf**. The value returned by **system** is the return status of the command executed.

For example, the following program:

```
$1 == "#include" { gsub(/[<>"]/, "", $2); system("cat " $2) }
```

calls the command **cat** to print the file named in the second field of every input record whose first field is "#include," after stripping any <, >, or " that might be present.

Co-operation with the shell

In all the examples thus far, the **awk** program is in a file and is retrieved using the **-f** flag, or it appears on the command line enclosed in single quotes, as in the following example:

```
awk '{ print $1 }' . . .
```

Since **awk** uses many of the same characters as the shell does, such as $ and ", surrounding the **awk** program with single quotes ensures that the shell passes the entire program unchanged to the **awk** interpreter.

Now, consider writing a command **addr** that searches a file *addresslist* for name, address, and telephone information. Suppose that *addresslist* contains names and addresses in which a typical entry is a multi-line record such as the following:

```
G. R. Emlin
600 Mountain Avenue
Murray Hill, NJ 07974
201-555-1234
```

Records are separated by a single blank line.

You want to be able to search the address list by issuing commands like the following:

```
addr Emlin
```

To do this, create a program of the form:

```
awk '
BEGIN              { RS = "" }
/Emlin/
' addresslist
```

The problem is how to get a different search pattern into the program each time it is run.

There are several ways to do this. One way is to create a file called *addr* that contains the following lines:

```
awk '
BEGIN                   { RS = "" }
/'$1'/
' addresslist
```

The quotes are critical here. The **awk** program is only one argument, even though there are two sets of quotes, because quotes do not nest. The **$1** is outside the quotes, visible to the shell, which then replaces it by the pattern *Emlin* when you invoke the command **addr Emlin**. On a UNIX system, you can make **addr** executable by changing its mode with the following command:

chmod +x addr

A second way to implement **addr** relies on the fact that the shell substitutes for $ parameters within double quotes:

```
awk "
BEGIN                    { RS = \"\" }
/$1/
" addresslist
```

Here you must protect the quotes defining **RS** with backslashes so that the shell passes them on to **awk**, uninterpreted by the shell. **$1** is recognized as a parameter, however, so the shell replaces it by the pattern when you invoke the following command:

addr ***pattern***

A third way to implement **addr** is to use **ARGV** to pass the regular expression to an **awk** program that explicitly reads through the address list with **getline**:

```
awk '
BEGIN   { RS = ""
          while (getline < "addresslist")
             if ($0 ~ ARGV[1])
                print $0
} ' $*
```

All processing is done in the **BEGIN** action.

Notice that you can pass any regular expression to **addr**; in particular, you can retrieve parts of an address or telephone number, as well as a name.

Example applications

awk has been used in surprising ways. We have seen **awk** programs that implement database systems and a variety of compilers and assemblers, in addition to the more traditional tasks of information retrieval, data manipulation, and report generation. Invariably, the **awk** programs are significantly shorter than equivalent programs written in more conventional programming languages, such as Pascal or C. In this section, we present a few more examples to illustrate some additional **awk** programs.

Generating reports

awk is especially useful for producing reports that summarize and format information. Suppose you want to produce a report from the file *countries,* that lists the continents alphabetically, and after each continent, its countries in decreasing order of population, like this:

```
     Africa:
                                Sudan          19
                                Algeria        18

     Asia:
                                China         866
                                India         637
                                USSR          262

     Australia:
                                Australia      14

     North America:
                                USA           219
                                Canada         24

     South America:
                                Brazil        116
                                Argentina      26
```

As with many data processing tasks, it is much easier to produce this report in several stages. First, create a list of continent-country-population triples, in which each field is separated by a colon. To do this, use the following program, **triples,** which uses an array **pop,** indexed by subscripts of the form 'continent:country' to store the population of a given country. The **print** statement in the **END** section of the program creates the list of continent-country-population triples that are piped to the **sort** routine:

```
     BEGIN  { FS = "\t" }
            { pop[$4 ":" $1] += $3 }
     END    { for (cc in pop)
                 print cc ":" pop[cc] | "sort -t: +0 -1 +2nr" }
```

The arguments for **sort** deserve special mention. The **-t:** argument tells **sort** to use : as its field separator. The **+0 -1** arguments make the first field the primary sort key. In general, **+i -j** makes fields **i+1, i+2, . . . , j** the sort key. If **-j** is omitted, the fields from **i+1** to the end of the record are used. The **+2nr** argument makes the third field, numerically decreasing, the secondary sort key (**n** is for numeric, **r** for reverse order). Invoked on the file *countries*, this program produces as output:

```
Africa:Sudan:19
Africa:Algeria:18
Asia:China:866
Asia:India:637
Asia:USSR:262
Australia:Australia:14
North America:USA:219
North America:Canada:24
South America:Brazil:116
South America:Argentina:26
```

This output is in the right order but the wrong format. To transform the output into the desired form, run it through a second **awk** program, **format**:

```
BEGIN  { FS = ":" }
{       if ($1 != prev) {
             print "\n" $1 ":"
             prev = $1
        }
        printf "\t%-10s %6d\n", $2, $3
}
```

This is a control-break program that prints only the first occurrence of a continent name and formats the country-population lines associated with that continent in the desired manner. The following command line produces the report:

```
awk -f triples countries | awk -f format
```

As this example suggests, complex data transformation and formatting tasks can often be reduced to a few simple **awks** and **sorts**.

Additional examples

This section gives some additional examples of how you might use **awk**.

Word frequencies

Our first example illustrates associative arrays for counting. Suppose you want to count the number of times each word appears in the input, where a word equals any contiguous sequence of non-blank, non-tab characters. The following program prints the word frequencies, sorted in decreasing order:

```
      { for (w = 1; w <= NF; w++) count[$w]++ }
END   { for (w in count) print count[w], w | "sort -nr" }
```

The first statement uses the array **count** to accumulate the number of times each word is used. Once the input has been read, the second **for** loop pipes the final count, along with each word, into the **sort** command.

Accumulation

Suppose you have two files of records, *deposits* and *withdrawals*, that contain a name field and an amount field. For each name, you want to print the net balance determined by subtracting the total withdrawals from the total deposits for each name. The net balance can be computed by the following program:

```
awk '
FILENAME == "deposits"    { balance[$1] += $2 }
FILENAME == "withdrawals" { balance[$1] -= $2 }
END                       { for (name in balance)
                            print name, balance[name]
} ' deposits withdrawals
```

The first statement uses the array **balance** to accumulate the total amount for each name in the file *deposits*. The second statement subtracts associated withdrawals from each total. If there are only withdrawals associated with a name, an entry for that name is created by the second statement. The **END** action prints each name with its net balance.

Random choice

The following function prints (in order) **k** random elements from the first **n** elements of the array **A**. In the program, **k** is the number of entries that still need to be printed, and **n** is the number of elements yet to be examined. The decision of whether to print the ith element is determined by the test **rand() < k/n**:

```
function choose(A, k, n) {
    for (i = 1; n > 0; i++)
        if (rand() < k/n--) {
            print A[i]
            k--
        }
    }
}
```

Shell facility

The following **awk** program simulates (crudely) the history facility of the UNIX system shell:

```
$1 == "=" { if (NF == 1)
          system(x[NR] = x[NR-1])
        else
          for (i = NR-1; i > 0; i--)
              if (x[i] ~ $2) {
                  system(x[NR] = x[i])
                  break
              }
        next }

/./      { system(x[NR] = $0) }
```

A line containing only = re-executes the last command executed. A line beginning with = *cmd* re-executes the last command whose invocation included the string *cmd*. Otherwise, the current line is executed.

Generating form-letters

The following program generates form letters:

```
BEGIN {                FS = "|"
                       while (getline <"form.letter")
          line[++n] = $0
}
{    for (i = 1; i <= n; i++) {
          s = line[i]
          for (j = 1; j <= NF; j++)
               gsub("\\$"j, $j, s)
          print s
     }
}
```

This program uses a template stored in a file called *form.letter*:

```
This is a form letter.
The first field is $1, the second $2, the third $3.
The third is $3, second is $2, and first is $1.
```

combined with replacement text of this form:

```
field 1 | field 2 | field 3
one | two | three
a | b | c
```

The **BEGIN** action stores the template in the array **template**; the remaining action cycles through the input data, using **gsub** to replace template fields of the form **$n** with the corresponding data fields.

In all such examples, a prudent strategy is to start with a small version and expand it, trying out each aspect before moving on to the next.

awk summary

The following sections summarize the functions and usage of **awk**.

Command line

awk *program filenames*
awk -f *program-file filenames*
awk -F *s* sets field separator to string *s*; **-Ft** sets separator to tab

Patterns

BEGIN
END
/regular expression
relational expression
pattern* && *pattern
pattern* | | *pattern
(*pattern*)
!pattern
pattern, pattern

Control flow statements

if (*expr*) *statement* [else *statement*]
if (*subscript* in *array*) *statement* [else *statement*]
while (*expr*) *statement*
for (*expr*; *expr*; *expr*) *statement*
for (*var* in *array*) *statement*
do *statement* while (*expr*)
break
continue
next
exit [*expr*]
return [*expr*]

Input-Output

close(***filename***)	close ***filename***
getline	set $0 from next input record; set NF, NR, FNR
getline < ***filename***	set $0 from next record of ***filename***; set NF
getline ***var***	set ***var*** from next input record; set NR, FNR
getline ***var*** < ***filename***	set ***var*** from next record of ***filename***
print	print current record
print ***expr-list***	print expressions
print ***expr-list*** > ***filename***	print expressions on ***filename***
printf ***fmt*** , ***expr-list***	format and print
printf ***fmt*** , ***expr-list*** > ***filename***	format and print on ***filename***
system(***cmd-line***)	execute command ***cmd-line***, return status

In **print** and **printf** above, >> ***filename*** appends to the ***filename***, and | ***command*** writes on a pipe. Similarly, ***command*** | **getline** pipes into **getline**. **getline** returns 0 on end of file, and -1 on error.

Functions

func ***name*** **(*parameter list*) { *statement* }**
function ***name*** **(*parameter list*) { *statement* }**
***function-name*(*expr*, *expr*, . . .)**

String functions

gsub(***r***, ***s***, ***t***)	substitute string ***s*** for each substring matching regular expression ***r*** in string ***t***, return number of substitutions; if ***t*** omitted, use $0
index(***s***, ***t***)	return index of string ***t*** in string ***s***, or 0 if not present
length(***s***)	return length of string ***s***
match(***s***, ***r***)	return position in ***s*** where regular expression ***r*** occurs, or 0 if ***r*** is not present
split(***s***, ***a***, ***r***)	split string ***s*** into array ***a*** on regular expression ***r***, return number of fields; if ***r*** omitted, FS is used in its place
sprintf(***fmt***, ***expr-list***)	print ***expr-list*** according to ***fmt***, return resulting string
sub(***r***, ***s***, ***t***)	like gsub except only the first matching substring is replaced
substr(***s***, ***i***, ***n***)	return ***n***-char substring of ***s*** starting at ***i*** ; if ***n*** omitted, use rest of ***s***

Arithmetic functions

atan2(***y***, ***x***)	arctangent of ***y***/***x*** in radians
cos(***expr***)	cosine (angle in radians)
exp(***expr***)	exponential
int(***expr***)	truncate to integer
log(***expr***)	natural logarithm
rand()	random number between 0 and 1
sin(***expr***)	sine (angle in radians)
sqrt(***expr***)	square root
srand(***expr***)	new seed for random number generator; use time of day if no ***expr***

Operators (Increasing precedence)

= += -= *= /= %= ^=	assignment
?:	conditional expression
||	logical OR
&&	logical AND
~ !~	regular expression match, negated match
< <= > >= != ==	relationals
blank	string concatenation
+ -	add, subtract
* / %	multiply, divide, mod
+ - !	unary plus, unary minus, logical negation
^	exponentiation (** is a synonym)
++ --	increment, decrement (prefix and postfix)
$	field

Regular expressions (Increasing precedence)

c	matches non-metacharacter ***c***
c	matches literal character ***c***
.	matches any character but newline
^	matches beginning of line or string
$	matches end of line or string
[***abc...***]	character class matches any of ***abc...***
[^***abc...***]	negated class matches any but ***abc...*** and newline
r1 | ***r2***	matches either ***r1*** or ***r2***
r1r2	concatenation: matches ***r1***, then ***r2***
r+	matches one or more ***r***'s
r*	matches zero or more ***r***'s
r?	matches zero or one ***r***'s
(***r***)	grouping: matches ***r***

Built-in variables

ARGC	number of command-line arguments
ARGV	array of command-line arguments (0.. **ARGC-1**)
FILENAME	name of current input file
FNR	input record number in current file
FS	input field separator (default blank)
NF	number of fields in current input record
NR	input record number since beginning
OFMT	output format for numbers (default **%.6g**)
OFS	output field separator (default blank)
ORS	output record separator (default newline)
RS	input record separator (default newline)
RSTART	index of first character matched by **match()**; 0 if no match
RLENGTH	length of string matched by **match()**; -1 if no match
SUBSEP	separates multiple subscripts in array elements; default "\034"

Limits

Any particular implementation of **awk** enforces some limits. The following limits exist in this implementation of **awk**: (Limits marked with an asterisk (*) are safe approximations.)

100 fields
3000* characters per input record
3000* characters per output record
3000* characters per individual field
3000* characters per printf string
400 characters maximum quoted string
250* characters in character class
55* open files or pipes
double precision floating point
Numbers are limited to what can be represented on the local machine, e.g., 1e-38..1e+38

Initialization, comparison, and type coercion

Each variable and field can potentially be a string or a number or both at any time. When a variable is set by the following assignment, its type is set to that of the expression:

```
var = expr
```

(Assignment includes +=, -=, etc.) An arithmetic expression is of type number, a concatenation is of type string, and so on. If the assignment is a simple copy, as in the following example, then the type of v1 becomes that of v2:

```
v1 = v2
```

In comparisons, if both operands are numeric, the comparison is made numerically. Otherwise, operands are coerced to string if necessary, and the comparison is made on strings. The type of any expression can be coerced to numeric by subterfuges such as:

```
expr + 0
```

and to string by:

```
expr ""
```

(that is, concatenation with a null string).

Uninitialized variables have the numeric value 0 and the string value "".

The type of a field is determined by context when possible; for example:

```
$1++
```

implies that $1 is to be numeric. The following implies that $1 and $2 are both to be strings:

```
$1 = $1 "," $2
```

Coercion is done as needed.

In contexts where types cannot be reliably determined, as in the following example, the type of each field is determined on input:

```
if ($1 == $2) ...
```

All fields are strings; in addition, each field that contains only a number is also considered numeric.

Fields that are explicitly null have the string value "" ; they are not numeric. Non-existent fields (i.e., fields past NF) are treated this way, too.

As it is for fields, so it is for array elements created by **split()**.

Mentioning a variable in an expression causes it to exist, with the value "" as described above. Thus, if **arr[i]** does not currently exist, the following expression causes it to exist with the value "" so the **if** is satisfied:

```
    if (arr[i] == "") ...
```

The following special construction determines if **arr[i]** exists without the side effect of creating it if it does not:

```
    if (i in arr) ...
```

Chapter 13

Using DOS accessing utilities

DOS tools are provided to help you bridge between the two operating systems. These tools are an extension of the features available on UNIX systems. These programs allow you, while working on your UNIX system, to access DOS files and directories which reside in a non-mounted DOS partition. If your system administrator permits, you can also gain access DOS files by mounting a DOS filesystem and using them directly. This chapter discusses both methods.

Accessing DOS files

The following is a list of the **dos**(C) commands and their functions:

doscat Copies one or more DOS files to the standard output. By default, the standard output is the terminal screen. If more than one file is specified, the files are displayed concatenated. **doscat** functions like the UNIX system **cat**(C) command. The following is an example of its usage:

```
doscat /dev/fd0:/john/memos
```

doscp Copies files between DOS and UNIX system environments. The first file specified is copied to the second file. Example of usage:

```
doscp /mary/list /dev/fd0:/budget/list
```

dosdir Lists DOS files in the standard DOS-style directory format. Example of usage:

```
dosdir /dev/fd0:/john
```

dosformat Creates a DOS 2.0 formatted diskette. Example of usage

```
dosformat /dev/fd0
```

dosls Lists DOS files and directories in a UNIX system format. Example of usage:

```
dosls /dev/fd0:/john
```

dosrm Removes files from a DOS disk. Example of usage:

```
dosrm /dev/fd0:/john/memos
```

dosmkdir Creates a directory on a DOS disk. Example of usage:

```
dosmkdir /dev/fd0:/john/memos
```

dosrmdir Deletes directories from a DOS disk. Example of usage:

```
dosrmdir /dev/fd0:/john/memos
```

Note that you must have a bootable, although not mounted, DOS partition on the hard disk or a DOS floppy in order to use these UNIX system commands. For example, you can only transfer a file from a UNIX system partition on the hard disk to a DOS floppy if either the DOS floppy is bootable or there is also a DOS partition on the hard disk. For more information about the DOS accessing utilities, refer to the *User's Reference.*

In order that the DOS utilities function correctly, the DOS partition/drive you are accessing must first be formatted using the DOS format command (i.e. the one in the DOS operating system). The volume label must also be set to a non-null value, and in DOS 4.0 or above the serial number must be set. Failure to comply with any of these actions may result in failure of the UNIX DOS utilities.

You might also be able to use the UNIX system **dd**(C) and **diskcp**(C) commands to copy and compare DOS floppies. The UNIX system **dtype**(C) command tells you what type of floppies you have (various DOS and UNIX system floppies).

In addition, the file */etc/default/msdos* describes which DOS filesystems (e.g. A:, B:, C:. . .) correspond to which UNIX system devices.

NOTE You cannot execute (run) DOS programs or applications under the UNIX operating system.

If you have the Development System, with the cmerge compiler, you can create and compile programs that can be run under DOS operating systems. Refer to the *DOS-OS/2 Development Guide* for more information. Also, see the DOS section in the *Programmer's Reference.*

Copying groups of files

The **doscp** command does not allow the use of wildcards, so it is only possible to copy one file at a time. To work around this restriction so that you can copy groups of files to or from a DOS diskette or partition, you must create the following shell scripts:

DOS file format to UNIX system file format transfer script:

```
: fromdos: copy a batch of files from DOS to UNIX format
if [ "$1" = "" ]
then
    echo "Usage: $0 disk:[/dospath/directory]"
    exit 1
fi
dosdir=$1
names=`dosls $dosdir`
for i in $names
do
    doscp "$dosdir/$i" `echo $i | tr "[A-Z]" "[a-z]"`
done
```

UNIX system file format to DOS file format transfer script:

```
: todos: copy a batch of files from UNIX format to DOS
if [ $# -lt 2 ]
then
    echo "Usage: $0 file [file ...] disk:[/dospath/directory]"
    exit 1
fi
files="$1"
while [ "$2" != "" ]
do
    files="$files $1"
    shift
done
dosdir=$1
for i in $files
do
    doscp $i $dosdir
done
```

You should create both of these scripts as executable files in the */usr/bin* directory and give them appropriate names such as "fromdos" or "todos." Use the **chmod**(C) command to make the files executable. For example:

chmod 755 /usr/bin/fromdos

This command gives read and execute permissions for the file *fromdos* to all users. Once you set these permissions, you can use the filename as a substitute for **doscp** on your command line, as in the following example:

fromdos /dev/fd0:/john

Using mounted DOS filesystems

In addition to using the DOS utilities provided with to manipulate DOS files, you can also mount a DOS filesystem and access its files directly while still operating from the UNIX system partition.

This means that you can edit or examine DOS files in place, without first copying them into the UNIX system filesystem. The major restriction is that DOS files and applications cannot be executed under this arrangement; this requires use of VP/ix (if operating from the UNIX system partition) or booting of the DOS partition. However, you can examine, copy, or edit data files and text files.

The operating system deals with DOS filesystems by superimposing certain qualities of UNIX system filesystems over the DOS filesystem without changing the actual files. UNIX system filesystems are highly structured and operate in a multiuser environment. In order to make DOS files readily accessible, access permissions and file ownership are superimposed on the DOS filesystem when mounted.

Mounting a DOS filesystem

Only *root* can mount a filesystem. Access by users is governed by the permissions and ownership that *root* places on the DOS filesystem. The system administrator must either mount the DOS filesystem or set up the system so that users can use the **mnt**(C) command.

Mounting a floppy disk

For example, if the system administrator permits it, you can mount DOS floppy disks, as in the following example using the 96tpi floppy mounted on */mnt*:

```
mnt -f DOS /dev/fd096 /mnt
```

Because of the limitations discussed earlier, DOS does not recognize permissions or ownership. When mounted from the UNIX system partition, the DOS files behave as follows:

- The permissions and ownership of the filesystem are governed by the mount point. For example, if *root* creates a mount point */X* with permissions of 777, all users can read or write the contents of the filesystem. If the mount point is owned by *root*, all files within the DOS filesystem and any created by other users are all owned by *root*.

- The permissions for regular files are either 0777 for readable/writable files or 0555 for read-only files. This preserves the consistency of the DOS filesystem. If a user can access the filesystem, the user is limited by the permissions available under the DOS directory structure. This permission is read-only or read-write. When a file is created, the permissions are based on the **umask** of the creator. For example, if the user's **umask** is 022, this generates files with permissions of 777.

File and directory arguments

The file and directory arguments for DOS files take the form:

device:name

where ***device*** is a UNIX system pathname for the special device file containing the DOS diskette or DOS partition, and ***name*** is a pathname to a DOS file or directory. For example:

/dev/fd0:/john/memos

indicates that the file *memos* is in the directory */john*, and that both are in the device file */dev/fd0* (the UNIX system special device file for the primary floppy drive). Arguments without *device:* are assumed to be UNIX system files.

User configurable default file

For convenience, the user configurable default file */etc/default/msdos* defines DOS drive names that you can use in place of UNIX system special device file pathnames. These are short forms that the system administrator can set up for DOS filesystems using the A:, B:, C: convention instead of, for example, */dev/fd096ds15*.

Appearance of DOS files

Because no attempt is made to change the nature of DOS files, the carriage return character (^M) is visible when editing a DOS file from the UNIX system partition. (UNIX system files only use a newline, while DOS files use a carriage return and a newline.) Thus when a DOS file (for example, *NUM.BAS*) that contains a series of numbers is opened using **vi**(C), it looks something like this:

```
1111 2222 3333 4444 5555^M
6666 7777 8888 9999 0000^M
1111 2222 3333 4444 5555^M
6666 7777 8888 9999 0000^M
1111 2222 3333 4444 5555^M
6666 7777 8888 9999 0000^M
~
~
~
~
~
~
~
~
~

"NUM.BAS" 6 lines, 100 characters
```

You can either ignore these numbers, or remove them with the **dtox/xtod**(C) commands.

NOTE If you remove the carriage returns in a DOS file, you must replace them to use the file under DOS.

The dtox and xtod commands

These commands are the easiest way to switch the end-of-line format. For example, the following commands convert the file *NUM.BAS* to and from the UNIX system file format, respectively:

dtox NUM.BAS > ***filename***
xtod NUM.BAS > ***filename***

Newline conversions with DOS utilities

When the **doscat**(C) and **doscp**(C) commands transfer DOS format text files to UNIX format, they automatically strip the ^M character. When text files are transferred to DOS, the commands insert a ^M before each linefeed character. Under some circumstances, the automatic newline conversions do not occur. The **-m** option ensures that the newline conversion is carried out. The **-r** option overrides the automatic conversion and forces the command to perform a true byte copy regardless of file type.

NOTE All DOS utilities leave temporary files in */tmp*, regardless of whether or not the utility executed successfully. These files are removed at the next reboot.

Other restrictions

This section explains the additional restrictions that you must observe.

Filenames

The rules for filenames and their conversion follows the guidelines found in the **dos**(C) manual page in the *User's Reference*. All DOS filenames have a maximum of eight characters, plus a three-character extension. For example, if you attempt to create a file named *rumplestiltski* within the DOS filesystem, it is truncated to *rumplest*:

In addition, the standard DOS restrictions on illegal characters apply. However, you can use wildcards just as you use them with UNIX system filesystems.

Modification times

When accessed from the UNIX system partition, the creation, modification, and access times of DOS files are always identical and use GMT, or Greenwich Mean Time. (This is because UNIX systems use GMT internally and convert it for the user.) This means that files created in the DOS filesystem will not have consistent times across the operating systems.

UNIX backup utilities

You cannot use the UNIX system **backup**(C) utility to make backups of a mounted DOS filesystem. However, DOS utilities and other copy programs like **tar**(C) work as expected.

For more information, including more technical aspects of DOS usage, refer to the **dos**(C) page in the *User's Reference.*

Summary

This chapter covers the following commands:

Table 13-1 Command review

Command	Description
doscat(C)	Copies one or more DOS files to the standard output
doscp(C)	Copies files between DOS and UNIX environments
dosdir(C)	Lists DOS files in the standard DOS-style directory format
dosformat(C)	Creates a DOS 2.0 formatted diskette
dosls(C)	Lists DOS files and directories in a UNIX format
dosrm(C)	Removes files from a DOS disk
dosmkdir(C)	Creates a directory on a DOS disk
dosrmdir(C)	Deletes directories from a DOS disk
dd(C)	Copies and compares DOS floppies
diskcp(C)	
dtype(C)	Determines the floppy type
chmod(C)	Changes the permissions on files and directories
mnt(C)	Mounts a filesystem
dtox(C)	Convert DOS file to UNIX file format
xtod(C)	Convert UNIX file to DOS file format

Appendix A

Sample shell startup files

This appendix contains sample listings and line-by-line explanations of the following shell startup files:

Bourne shell (**sh**)	*.profile*
Korn shell (**ksh**)	*.profile* *.kshrc*
C shell (**csh**)	*.login* *.cshrc*

Line numbers have been added to all the file listings for purposes of explanation; numbers do not appear in the actual files.

The Bourne shell .profile

The Bourne shell (**sh**) reads a single file in your home directory, the *.profile*. A typical Bourne shell *.profile* might look something like this:

```
 1 :
 2 #       @(#) profile 23.1 91/04/03
 3 #
 4 # .profile       -- Commands executed by a login Bourne shell
 5 #
 6 # Copyright (c) 1985-1991, The Santa Cruz Operation, Inc.
 7 # All rights reserved.
 8 #
 9 # This Module contains Proprietary Information of the Santa Cruz
10 # Operation, Inc., and should be treated as Confidential.
11 #
```

```
12 PATH=$PATH:$HOME/bin:.                    # set command search path
13 MAIL=/usr/spool/mail/'logname'            # mailbox location
14 export PATH MAIL

15 # use default system file creation mask

16 eval 'tset -m ansi:ansi -m :?${TERM:-ansi} -r -s -Q'
```

line 1 contains a single colon that says "execute this script as a Bourne shell script." This is a convention for scripts written in Bourne shell, so C shells know to start a new **sh** to run the Bourne shell scripts.

(C shells need to start Bourne shells to run Bourne shell scripts because they do not understand the Bourne shell language. The Korn shell, however, is compatible with the Bourne shell, so you can use most Bourne shell scripts in the Korn shell without a problem.)

lines 2-11 contain comments. Each line that starts with a # (number sign) is a comment. The shell ignores these lines. In this case, lines 2-11 contain SCO copyright information.

line 12 sets the path. It says, "set the path equal to the current path, plus the *bin* in the home directory, plus the current directory (.)." Setting the path to the existing path presumes there is a system-wide */etc/profile* that sets up a path definition for all users. The path definition in */etc/profile* would contain the usual command directories, such as */bin* and */usr/bin.*

line 13 tells the shell where to find mail. The `'logname'` in backquotes tells the shell to substitute the output of the command **logname**(C), which returns a user's login name. Because `'logname'` is used instead of a particular login name, this script works for any user.

line 14 tells the shell to export the **PATH** and **MAIL** settings to all its sub shells. This guarantees that if you type **sh** to start a new Bourne shell, the new Bourne shell has the same path definition and mail setup as your login Bourne shell.

line 15 contains a comment, like lines 2-11. This comment tells us that login Bourne shells use the default system file creation mask, which is set in */etc/profile.* This explains why there is no **umask** setting in this *.profile.*

line 16 sets up the terminal type, using the **tset**(C) (terminal setup) command. **tset** sets your terminal type, as well as the erase and kill characters for your terminal.

This **tset** command says "check if this serial line is mapped to *ansi* in the */etc/ttytype* file; if it is, set the terminal type to *ansi*. Otherwise, prompt the user with `TERM:ansi`." The **-r** option prints the terminal type on the screen, **-s** exports the terminal type to any subshells, and **-Q** suppresses the `Erase set to ...`, `Kill set to ...` messages that **tset** would otherwise show. The **tset** command is enclosed in backquotes and preceded by the shell command **eval** to guarantee that all necessary substitutions are made within the **tset** command before it is evaluated by the shell.

The Korn shell .profile and .kshrc

The Korn shell uses two startup files, the *.profile* and the *.kshrc*. The *.profile* is read once, by your login **ksh**, while the *.kshrc* is read by each new **ksh**.

A typical Korn shell *.profile* might look something like this:

```
 1 :
 2 #       @(#) profile 23.1 91/04/03
 3 #
 4 # .profile       -- Commands executed by a login Korn shell
 5 #
 6 # Copyright (c) 1990, 1991, The Santa Cruz Operation, Inc.
 7 # All rights reserved.
 8 #
 9 # This Module contains Proprietary Information of the Santa Cruz
10 # Operation, Inc., and should be treated as Confidential.
11 #

12 PATH=$PATH:$HOME/bin:.                  # set command search path
13 export PATH

14 if [ -z "$LOGNAME" ]; then
15         LOGNAME=`logname`               # name of user who logged in
16         export LOGNAME
17 fi
```

```
18 MAIL=/usr/spool/mail/$LOGNAME           # mailbox location
19 export MAIL

20 if [ -z "$PWD" ]; then
21         PWD=$HOME                       # assumes initial cwd is HOME
22         export PWD
23 fi

24 if [ -f $HOME/.kshrc -a -r $HOME/.kshrc ]; then
25         ENV=$HOME/.kshrc        # set ENV if there is an rc file
26         export ENV
27 fi

28 # use default system file creation mask (umask)

29 eval `tset -m ansi:ansi -m $TERM:?${TERM:-ansi} -r -s -Q`

30 # If job control is enabled, set the suspend character to ^Z (control-z):
31 case $- in
32 *m*)    stty susp '^z'
33         ;;
34 esac

35 set -o ignoreeof                # don't let control-d logout

36 case $LOGNAME in                # include command number in prompt
37 root)   PS1="!# " ;;
38 *)      PS1="!$ " ;;
39 esac
40 export PS1

41 /tcb/bin/prwarn                 # issue a warning if password due to expire
```

line 1	contains a single colon that says "this is a Bourne shell script." Even though this is a startup script for the Korn shell, the authors have chosen to use the more common syntax of the Bourne shell programming language.
lines 2-11	contain comments.
line 12	sets the path definition in exactly the same way as the preceding Bourne shell *.profile*: "set the path equal to the current path, the *bin* in the home directory, and the current directory."
line 13	exports the path to any subshells. This way, you do not have to include a path definition in your *.kshrc*.

lines 14-17	set up a variable called **LOGNAME**, which is used in the following **MAIL** setting (line 18). Literally, these lines say "if checking for the value of **LOGNAME** returns a zero-length string (that is, if **LOGNAME** is not set), then set **LOGNAME** to the output of the **logname** command. Then, export the **LOGNAME** variable to all subshells."
line 18	tells the shell where to look for mail, using the variable **LOGNAME**.
line 19	exports the mail location to all subshells.
lines 20-23	check to see if a variable is already set, and if it is not, set the variable. These lines are similar to lines 14-17. In this case, **PWD** is being set to the home directory.
lines 24-27	check for a *.kshrc* file in the home directory, and set the **ksh** variable **ENV** to this file if it exists. **ksh** looks in the file pointed to by the **ENV** variable to set up the environment for every new **ksh**; you need to tell it explicitly to look in *~/.kshrc* for **ENV** definitions. Literally, these lines say "if a file called *.kshrc* exists in the home directory and the file is readable, then set **ENV** to point to this file, and export **ENV**."
line 28	contains a comment. Just as in the preceding Bourne shell *.profile*, **umask** is not set here. The authors have chosen to use the default system **umask** rather than resetting it on a per-user basis.
line 29	sets up the terminal type using **tset**(C), as explained in the preceding Bourne shell *.profile*.
lines 30-34	test to see if job control is enabled, and if it is, set the suspend character to ⟨Ctrl⟩**z**. Job control is a Korn shell feature that lets you move jobs you are processing from the foreground to the background and vice versa. You use the suspend character to suspend a job temporarily that is running in the background.
line 35	tells the shell to ignore single end-of-file (EOF) characters. This is what you set to stop ⟨Ctrl⟩**d** from logging you out.
lines 36-40	set up the prompt based on the value of **LOGNAME**. For normal users, the prompt is the current command number followed by a " $ "; for *root* (the super user), the prompt is the current command number followed by a " # ".
line 41	runs the command **prwarn**(C), which warns you if your password is due to expire soon.

A typical *.kshrc* might look like this:

```
 1 :
 2 #
 3 # .kshrc -- Commands executed by each Korn shell at startup
 4 #
 5 #        @(#) kshrc 1.1 90/03/13
 6 #
 7 # Copyright (c) 1990, The Santa Cruz Operation, Inc.
 8 # All rights reserved.
 9 #
10 # This Module contains Proprietary Information of the Santa Cruz
11 # Operation, Inc., and should be treated as Confidential.
12 #

13 # If there is no VISUAL or EDITOR to deduce the desired edit
14 #  mode from, assume vi(C)-style command line editting.
15 if [ -z "$VISUAL" -a -z "$EDITOR" ]; then
16         set -o vi
17 fi
```

line 1	tells the shell that this is a Bourne shell script by starting the script with a single colon, as you have seen before.
lines 2-14	contain comments. These make up the bulk of this brief *.kshrc*.
lines 15-17	set up **vi**(C) as the default editor **ksh** uses when you want to edit a command line. Literally, these lines say "If the **VISUAL** variable is not set, and the **EDITOR** variable is not set, then turn on (**set -o**) the **vi** option."

The C-shell .login and .cshrc

The C shell, like the Korn shell, uses one file to set up the login environment and a different file to set up environments for every subsequent C shell. In C shell, *.login* is the file read only at login, and *.cshrc* is the file read each time a **csh** is started.

While both the Bourne shell and the Korn shell use Bourne shell startup scripts, the C shell uses C-shell startup scripts, so you will notice that variables are set and tests are performed slightly differently. C-shell scripts do not start with a " : " because they are intended for use with C shells, not Bourne shells.

A typical C-shell *.login* might look something like this:

```
 1 #       @(#) login 23.1 91/04/03
 2 #
 3 # .login-- Commands executed only by a login C-shell
 4 #
 5 # Copyright (c) 1985-91, The Santa Cruz Operation, Inc.
 6 # All rights reserved.
 7 #
 8 # This Module contains Proprietary Information of the Santa Cruz
 9 # Operation, Inc., and should be treated as Confidential.
10 #

11 setenv SHELL /bin/csh

12 set ignoreeof                          # don't let control-d logout
13 set path = ($path $home/bin .)    # execution search path

14 set noglob
15 set term = (`tset -m ansi:ansi -m :?ansi -r -S -Q`)
16 if ( $status == 0 ) then
17         setenv TERM "$term"
18 endif
19 unset term noglob

20 /tcb/bin/prwarn         # issue a warning if password due to expire
```

lines 1-10	contain comments.
line 11	sets the environment variable **SHELL** to be */bin/csh*.
line 12	tells **csh** to ignore single end-of-file (EOF) characters; in other words, do not let ⟨Ctrl⟩**d** log out, as the comment says.
line 14	turns on the **noglob** setting. The **noglob** setting, which prevents filename expansion, is turned on before a **tset**(C) command is attempted (on line 15). Without **noglob**, the **tset** command would be read incorrectly.
lines 15-19	set up your terminal type using **tset**. Line 15 is the **tset** command you have seen before. Line 16 tests to make sure the **tset** command succeeded and, if it did, line 17 sets the environment variable **TERM**. Line 18 closes the **if** statement. Line 19 unsets the **term** variable and turns off **noglob**, so filenames now expand as expected when wildcard characters are used.
line 20	runs **prwarn** to warn you if your password is due to expire.

A typical *.cshrc* might look like this:

```
 1 #
 2 # .cshrc -- Commands executed by the C-shell each time it runs
 3 #
 4 #       @(#) cshrc 3.1 89/06/02
 5 #
 6 # Copyright (c) 1985-1989, The Santa Cruz Operation, Inc.
 7 # All rights reserved.
 8 #
 9 # This Module contains Proprietary Information of the Santa Cruz
10 # Operation, Inc., and should be treated as Confidential.
11 #

12 set noclobber                              # don't allow '>' to overwrite
13 set history=20                             # save last 20 commands
14 if ($?prompt) then
15         set prompt=%               # set prompt string
16 # some BSD lookalikes that maintain a directory stack
17         if (! $?_d) set _d = ()
18         alias   popd    'cd $_d[1]; echo ${_d[1]}:; shift _d'
19         alias   pushd   'set _d = (`pwd` $_d); cd *'
20         alias   swapd   'set _d = ($_d[2] $_d[1] $_d[3-])'
21         alias   flipd   'pushd .; swapd ; popd'
22 endif
23 alias print 'pr -n :* | lp'                # print command alias
```

lines 1-11	contain comments.
line 12	turns on **noclobber**, which prevents you from unintentionally overwriting files using output redirection.
line 13	sets the length of the command history to 20 commands. Both **ksh** and **csh** keep track of old commands and allow you to re-use them.
lines 14-15	check to see if the prompt string is set, and, if it is not, set it to be a "%".
lines 16-22	set up some command aliases to perform directory stack manipulation. These commands are familiar to users of the Berkeley (Berkeley Standard Distribution — BSD) UNIX system.
line 23	sets up the **print** alias, which runs files through the **pr**(C) print program before sending them to the printer.

Index

A

Abbreviating in vi, 52
ADM manual pages, 5
Alias
  in C shell, 202, 204-205, 207
  in Korn shell, 238
  in mail, 80
Alphabetizing, 56
Appending
  in Bourne shell, 146
  in C shell, 204
  in vi, 23
at command, 110
audittrail authorization, 132
Authorizations, 131
auths command, 133
awk
  actions, 270, 278-279
  arguments, command-line, 300
  arithmetic, 279
  arrays, 289
  comments, 292
  debugging, 272
  examples, 271, 302, 305-307
  fields, 267
  flow control, 287
  functions
    string, 282-284
    user-defined, 291
  functions, 271
  input
    separators, 297
  input, 296-297
  lexical conventions, 292
  limits, 311
  multi-line records, 297
  operators, 274
  output
    formatted, 269, 293-294
    printing, 292
    separators, 293
  output, 268, 292, 295, 303
  patterns
    BEGIN and END, 273

awk *(continued)*
  patterns *(continued)*
    combining, 277
    ranges, 278
    regular expressions, 275
    relational expressions, 274
  patterns, 269, 273
  program structure, 266
  strings, 282
  summary, 307-312
  using with shell, 300-301
  variables
    built-in, 271, 278
    field, 285
    type, 286
    user-defined, 271
awk, 266

B

Background processing
  in C shell, 204
  in Korn shell, 232, 234
  process ID (PID) in Bourne shell, 155
Background processing, 101, 160-161
Backspace key, 6
backup command, 322
Backups
  creating, 94-95
  extracting, 96
  listing contents, 95
  shorthand tar notation, 98
Backups, 91, 93
batch command, 108
/bin, 143
Bourne shell (sh)
  See also Shell.
  arguments
    expanding, 146, 159
    number of, 154
    rescanning, 159
    testing, 155
  background processing, 160
  command

Bourne shell (sh) *(continued)*
  command *(continued)*
    built-in commands, 173, 184
    exit status, testing, 154
    grouping, 166
    substitution, 144, 149
    supporting commands, 177, 179-180, 184
  flags, 156, 177, 183
  input/output, 146, 168, 180-181
  invoking, 158
  process ID number (PID), 154-155
  .profile, 153, 156, 169, 323
  programming
    argument passing, 158
    arithmetic, 179
    debugging, 156, 177
    flow control, 161-166
    functions, 167
    in-line input documents, 180, 190
    interrupt handling, 169-171
    reading in a file, 169
    summary, 194
  programming, 183
  prompt ($), 48
  quoting, 144, 149, 151
  redirection, 146, 148
  variables
    assignment, 149-151
    conditional substitution, 181
    exporting, 153, 157
    keyword parameter, 177
    parameter substitution, 151
    positional parameters, 150, 154, 158-159, 182
    pre-defined, 154
    substitution, 151
    user-defined, 150-151
  variables, 145, 150, 153, 157
Bourne shell (sh), 141
Buffers
  in kernel, 143
  in vi, 26

C

C manual pages, 5
C shell (csh)
  See also Shell.
  argument
    expansion, 211, 217
  background processing, 204-205
  command
    adding, 199
    alias, 202, 204-205, 207
    built-in commands, 205-206
    executing, 199
    exit status, testing, 210
    history, 200-202
    repeating, 205
    separator (;), 203
    substitution, 213, 218
  .cshrc, 197, 323
  flags, 216
  input/output, 204, 215, 219
  invoking, 197
  logging out, 198
  .login, 197, 323
  path variable, 198
  programming
    colon (:) modifiers, 213
    comments, 207, 213
    debugging, 216
    example, 211
    execution, 207, 216
    flow control, 211-212, 214, 216
    inline data, 215
    interrupt handling, 215
    operators, 210
    reading in a file, 206
  programming, 207, 209
  prompt (%), 48
  quoting, 203, 216-217, 219
  redirection, 204, 219
  special characters, 218-219
  variables
    substitution, 209
    accessing, 208
    assignment, 198
    built-in, 199-200
    expansion, 217
    setting, 206
    substitution, 208
  variables, 198
C shell (csh), 197
Calendar reminder service, 82
Calling a remote system, 121, 123
cancel command, 54

Canceling
  mail message, 72
  vi session, 19
cat command, 102
cd command, 155, 224, 231
CDPATH variable, 153
Changing
  directories, 155, 231
  display in vi, 18
  passwords, 129
Characters
  counting, 58
  typing ahead, 143
chmod command, 317
chmodsugid authorization, 132
chown command, 132
Command
  See also individual command names.
  alias
    in C shell, 202, 204-205, 207
    in Korn shell, 238
  command list, 160
  editing
    in Korn shell, 224
  editing, 6
  entering, 6
  environment, 157
  execution
    remote, 120
  execution, 101, 142-143, 186
  exit status, testing in Bourne shell, 154
  naming, 143
  repeating
    in C shell, 205
    in Korn shell, 229
    in vi, 10, 12
  substitution, 144, 149
  undoing, in vi, 10
Command, 160
Comparing files, 56
Control characters, inserting in vi, 28
Copying
  DOS files, 317
  text in vi
    See also vi, yanking.
  text in vi, 26
Core file, 6
Correcting mistakes in vi, 24
Counting words, lines, characters, 58
CP manual pages, 5
Crash recovery in vi, 46
cron command, 106
crontab command, 106
crypt command, 137
.cshrc file, 197, 323
ct command, 121-123
cu command, 121, 123-125
Cursor, moving in vi, 11, 20
Customizing
  mail, 80
  vi, 53

D

dd command, 316
Debugging
  See also individual shells and awk.
  creating core file, 6
Decryption, 138
Delete key, 6
Deleting
  in vi, 12, 28, 60
  mail, 71
Diagnostic output
  in Bourne shell, 146, 181
  in C shell, 204
diff command, 56
Directories
  changing, 155, 231
  naming, 144, 176
  searching, 143, 153, 186-187, 198
  size, 187
diskcp command, 99, 316
Disks
  copying, 99
  extracting file from, 93
  formatting, 92
  listing contents, 93
  number needed, 94
  reporting usage, 205
  saving file to, 93
Disks, 91
Display, changing, in vi, 18
DOS
  accessing DOS files, 315
  appearance of files, 320
  backups, 322
  cross-compiling, 316
  doscat command, 315, 321

DOS *(continued)*
doscp command, 315, 317, 321
dosdir command, 315
dosformat command, 315
dosls command, 316
dosmkdir command, 316
dosrm command, 316
dosrmdir command, 316
/etc/default/msdos file, 316, 319
file and directory arguments, 319
filenames, 321
filesystems, mounting, 318
manual pages, 5
utilities, newline conversions, 321
dtox command, 320
dtype command, 316
du command, 94, 205

E

echo command, 179
ed scripts, 253
Editing, command line, 6
Editors, *See also* sed and vi.
Editors, 7
Electronic mail (e-mail). *See* mail.
Encryption
commands, 136
editing encrypted files, 137
Encryption, 137
End-of-line, displaying, in vi, 49
Enter key, 6
Environment, customizing, in mail, 80
Environment, 157
Environment, customizing
in mail, 80
in vi, 53
Erase key, 6
Escape key
in vi, 9, 52
Escape key, 6
execsuid authorization, 132
EXINIT variable, 47
Exiting
mail, 73, 77
shell, 6
vi, 17, 41, 44
expr command, 179
.exrc file, 53

F

F manual pages, 5
false command, 180
Field separator, in Bourne shell, 152
Files
comparing, 56
creating in vi, 8
DOS files, accessing, 315
encryption/decryption, 137-138
file descriptor, 146, 180
filename
argument, 143
DOS filenames, 321
finding text in, 57
naming in vi, 43
overwriting in C shell, 200
printing, 53
saving to disk or tape, 93
security, 136
sorting, 56
transferring
to non-UNIX systems, 123
with cu, 125
with uucp or uuto, 114
Filter, 147
Finding text in a file, 57
Form letters, 307
format command, 91
Formatting
disks, 92
tapes (mini cartridge), 93
FP manual pages, 5

G

Global substitution, 253
grep command, 57, 109

H

Help
in mail, 72, 78
online manual pages, 4
HOME variable, 152
HW manual pages, 5

I

IFS variable, 152
Input. *See* Standard input or individual shells.
Inserting text in vi, 9, 23-24

K

K manual pages, 5
Kernel authorizations, 131
Keys
  mapping, 52
  modifying, 6
Keys, 6
Keyword parameter, 157
kill command, 102, 104, 205, 235
Killing
  process, 102, 104
Korn shell (ksh)
  background processing, 232-234
  Bourne shell compatibility, 221, 246
  command
    alias, 238
    built-in commands, 244
    execution, 237
    history, 228-230
    interpretation, 237
    repeating, 229
    substitution, 241
  command-line editing, 224-230
  editing command line, 225
  expanded cd capabilities, 231
  flags, 223
  invoking
    with filename, 222
  invoking, 222
  job control, 221, 232-235
  .kshrc, 323
  parentheses, 241
  pattern matching, 251
  programming
    debugging, 251
    dot scripts, 246
    functions, 246-247
    substring parameter expansion, 248
    user input, 250
  programming, 222, 242
  prompt ($), 48
  quoting, 240-241
Korn shell (ksh) *(continued)*
  tilde expansion, 241
  variables, environment, 224
Korn shell (ksh), 221
.kshrc file, 323

L

Lines
  breaking in vi, 28
  counting, 58
  current line in vi, 9
  joining in vi, 28
Locked account/terminal, 129
Logging in, 135
Logging out, in C shell, 198
Login
  changing shell, 222
  directory, 152
  ending session, 6, 166
  remote login, 121, 124
  security, 128
  spoofing program, 128
.login file, 197, 323
logout command, 198
lp command, 53
lpstat command, 55

M

M manual pages, 5
mail
  alias, 80
  canceling, 72
  carbon copy, 73, 79, 81
  command
    summary, 88
    syntax, 69
  compose escapes, 67, 78
  customizing, 80
  deleting, 71, 75
  editing
    with vi or ed, 68
  editing, 71
  folder, 74, 87
  forwarding
    automatic, 86
  forwarding, 76
  headers

mail *(continued)*
  headers *(continued)*
    editing, 78
    paging through, 69
  headers, 68-69, 72
  help, 72, 78
  mailbox
    mbox, 66
    system mailbox, 66
  .mailrc, 80
  message list
    special characters, 70
  message list, 70
  messages
    listing in chronological order, 81
  messages, 66
  options
    carbon copy (-c), 68
    subject (-s), 68
  options, 68
  organizing, 87
  printing, 75
  prompt, 72
  quitting
    with no changes, 77
  quitting, 73, 77
  reading, 70, 72
  reading in
    dead letter, 79
    file, 79
    message, 79
  reminder service, 82
  replying to, 75, 79
  saving
    automatic, 75, 77
  saving, 69, 71, 73-74, 81
  screen size, setting, 69
  sending
    to remote sites, 76-77
  sending, 71-72
  shell escape, 76
  subject, 68, 71, 73
  system defaults, 80
  undeleting, 75
mail, 65
MAIL variable, 152
MAILCHECK variable, 152
MAILPATH variable, 152
.mailrc file, 80
makekey command, 137
man command, 4
Manual pages, 4-5
Mapping keys, 52
Matching. *See* Pattern matching.
mcart command, 93
mem authorization, 132
Messages in vi, 51
Metacharacters. *See* Special characters.
Mistakes, correcting in vi, 24
mnt command, 318
Mode, changing, 6
Modems, calling a remote system, 121, 123
Mounting DOS filesystems, 318

N

nice command, 105, 232, 236
nohup command, 160, 232, 236

O

Output. *See* Standard output or individual shells.
Overwriting, file, in C shell, 200

P

Password
  changing, 129
  security, 134
  warn of imminent expiration, 326
Pasting text in vi. *See* vi, putting.
Path. *See* Shell, search path.
PATH variable, 143, 153, 186
Pattern matching
  See also awk.
  in Korn shell, 251
  in shell, 143-144, 146, 151
  in vi, 14, 35, 38-39
Pipe, 147, 160, 204
printerstat authorization, 132
Printing
  canceling, 54
  status, 55
Printing, 53
printqueue authorization, 132
Process
  See also Background processing.

Process *(continued)*
  background, 101
  delaying execution, 105
  ID number (PID)
    in Bourne shell, 154-155
    listing, 205
  killing, 102, 104
  prioritizing, 105
  scheduling, 106
  status, 102
  terminating, 235
Process, 142
.profile file, 153, 156, 169, 323
Programming. *See* Shell, individual shells, and awk.
Prompt
  colon (:), 16
  dollar sign ($), 48
  percent sign (%), 48
  PS1 variable, 153
  PS2 variable, 153
  question mark (?) in mail, 72
Protected subsystem, 128
prwarn command, 326
ps command, 102, 205

Q

queryspace authorization, 132
Quoting
  in Bourne shell, 151
  in C shell, 203, 216-217, 219
  in Korn shell, 240-241
  with uux command, 121
Quoting, 144, 149

R

Redirection
  in Bourne shell, 146, 148, 168
  in C shell, 204, 219
Reference manuals. *See* Manual pages.
Regular expressions, *See also* pattern matching.
Regular expressions, 254, 310
Reminder service, 82
Remote login. *See* Login, remote login.
Repeating command
  in C shell, 205-206
Repeating command *(continued)*
  in Korn shell, 229
  in vi, 10, 12, 41
Return code. *See* Command, exit status.
Return key, 6
rm command, 102

S

S manual pages, 5
Saving file
  in vi, 41
  to disk or tape, 93
Scheduling processes, 106
Screen, redrawing, 37, 46
Scrolling, in vi, 11
Scrolling, 6
Searching, 14, 16, 36, 50
Secondary authorizations, 132
Security
  files, 136
  passwords, 134
  practices, 134
  user accounts, 135
Security administrator, 128
sed
  = function, 264
  { function, 264
  : label function, 264
  a function, 258
  addressing, 255
  b label function, 264
  B!function, 264
  c function, 258
  d function, 257
  D function, 262
  -e, 254
  -f, 254
  flow-of-control, 254
  flow-of-control functions, 264
  functions, 257
  g function, 260, 263
  G function, 263
  h function, 263
  H function, 263
  hold and get functions, 263
  i function, 258
  input/output functions, 261
  miscellaneous functions, 264

sed *(continued)*
  multiple input-line functions, 262
  -n, 254
  n function, 257
  N function, 262
  p function, 260-261
  P function, 262
  q function, 264
  r function, 261
  s function, 259
  substitution functions, 259
  t label function, 264
  w function, 260-261
  x function, 263
sh command. *See* Bourne shell (sh).
Shell
  See also individual shells.
  command execution, 142-143
  creating, 142
  escape
    in mail, 76
    in vi, 18, 44
  exiting, 6, 166
  identifying, 48
  input/output, 146
  PATH variable, 143
  pattern matching, 143-144, 146, 151
  programming
    examples, 187
    optimizing, 184
  programming, 142, 175
  search path, 143, 153
  startup files, 323
SHELL variable, 152
sleep command, 105
sort command, 56
Special characters
  in C shell, 218-219
  in mail message list, 70
  in vi, 39, 51
Special characters, 143
Spoofing program, 135
Standard error. *See* Diagnostic output.
Standard input, in Bourne shell, 146
Standard output, in Bourne shell, 146
Status
  line in vi, 15, 45
  process, 102
Sticky directories, 134
String
String *(continued)*
  abbreviating, in vi, 52
  searching for, in vi, 15
  variable, in Bourne shell, 150
stty command, 6
su authorization, 132
su command, 131
Subsystem authorizations, 132
sysadmsh command, 222

T

Tabs, displaying, in vi, 49
Tapes
  extracting file from, 93
  formatting, 93
  listing contents, 93
  saving file to, 93
Tapes, 91
tar command, 93, 95, 322
Telecommunication. *See* UUCP.
TERM variable, 48
terminal authorization, 132
Terminal type
  displaying in vi, 50
  setting, 48, 325
test command, 177
time command, 184, 206
Transferring files
  to non-UNIX systems, 123
  with cu, 125
Trojan horse, 127
true command, 180
Trusted computing base (TCB), 128
tset command, 325

U

Undoing command
  in vi, 10, 40
Users, currently on system, 57
/usr/bin, 143
UUCP
  connected sites, 76
  dial-out site, 114
  file permissions, 115
  listing remote UUCP systems, 115
  status of transfer, 118
  transferring files

UUCP *(continued)*
  transferring files *(continued)*
    indirect transfers, 115, 117
  transferring files, 114
  uucico daemon, 119
  uucp command, 116-117
  uuname command, 76, 115
  uupick command, 119-120
  uustat command, 118
  uuto command, 118-119
  uux command, 120-121
UUCP, 114

V

Variables. *See* shells and individual variables.
vi
  abbreviating, 52
  appending, 23-24
  buffers
    delete buffers, 33
    named buffers, 25
  buffers, 26
  colon (:) prompt, 16
  commands
    summary, 59, 63
  commands, 16, 45
  copying
    See also yanking.
  copying, 26
  crash recovery, 46-47
  cursor movement, 11, 20-23
  customizing, 53
  deleting, 12, 28-30, 33
  display, changing, 18
  editing several files, 43
  encryption, 137
  end-of-line, displaying, 49
  entering, 8, 20, 42
  error messages, 45, 50
  escape key, 9
  escaping to shell command line, 18
  EXINIT variable, 47
  .exrc file, 53
  file
    amount displayed, 15
    creating, 8, 20
    current (%), 52
    name, 45

vi *(continued)*
  file *(continued)*
    naming, 43
    saving, 41
    status, 15, 45
    switching, 44
  goto command (G), 11
  inserting, 9, 17, 23-24, 28
  joining lines, 28
  leaving, 19
  line
    addressing, 29
    breaking , 28
    current, 9
    moving to specific (G), 11
    numbers, 19, 26, 46, 50
  mail, using in, 67
  mapping, 52
  marking, 25
  mistakes, correcting
    See also deleting.
  mistakes, correcting, 24
  modes, 9, 46
  moving text, 32
  options, 18-19, 36, 39, 45-47, 49-51
  overwriting characters, 30
  pasting. *See* putting.
  problem solving, 46
  putting, 25
  quitting, 17, 41, 44
  redrawing screen, 37, 46
  repeating command, 10, 12, 41
  replacing
    choosing replacement, 38
    line, 31
    printing replacements, 37
    word, 31-32, 37
  searching
    backward, 35
    case significance, 36, 50
    forward, 14, 35
    matching, 38-39
    next match, 15, 36
    search and replace, 16, 36-37
    special characters, 36, 51
    wrap, 15, 36, 51
  shell escape, 44, 46
  status line, 15, 45
  substituting, 31-32
  tabs, displaying, 49

vi *(continued)*
  undo command, 10, 40
  yanking, 25, 27
vi, 7

W

wc command, 58
who command, 57
Wildcard characters. *See* Special characters.
Words, counting, 58

X

xtod command, 320

SCO®
OPEN SYSTEMS SOFTWARE

Please help us to write computer manuals that meet your needs by completing this form. Please post the completed form to the Technical Publications Research Coordinator nearest you: The Santa Cruz Operation, Ltd., Croxley Centre, Hatters Lane, Watford WD1 8YN, United Kingdom; The Santa Cruz Operation, Inc., 400 Encinal Street, P.O. Box 1900, Santa Cruz, California 95061, USA or SCO Canada, Inc., 130 Bloor Street West, 10th Floor, Toronto, Ontario, Canada M5S 1N5.

Volume title: ______________________________

(Copy this from the title page of the manual, for example, SCO UNIX Operating System User's Guide)

Product: ______________________________

(for example, SCO UNIX System V Release 3.2 Operating System Version 4.0)

How long have you used this product?

❑ Less than one month ❑ Less than six months ❑ Less than one year

❑ 1 to 2 years ❑ More than 2 years

How much have you read of this manual?

❑ Entire manual ❑ Specific chapters ❑ Used only for reference

	Agree				*Disagree*
The software was fully and accurately described	❑	❑	❑	❑	❑
The manual was well organized	❑	❑	❑	❑	❑
The writing was at an appropriate technical level (neither too complicated nor too simple)	❑	❑	❑	❑	❑
It was easy to find the information I was looking for	❑	❑	❑	❑	❑
Examples were clear and easy to follow	❑	❑	❑	❑	❑
Illustrations added to my understanding of the software	❑	❑	❑	❑	❑
I liked the page design of the manual	❑	❑	❑	❑	❑

If you have specific comments or if you have found specific inaccuracies, please report these on the back of this form or on a separate sheet of paper. In the case of inaccuracies, please list the relevant page number.

May we contact you further about how to improve SCO UNIX documentation? If so, please supply the following details:

Name ____________________ *Position* ____________________

Company ______________________________

Address ______________________________

City & Post/Zip Code ______________________________

Country ______________________________

Telephone ____________________ *Facsimile* ____________________

31 January 1992

AU01210P000