Sample **.bash_profile** and **.bashrc** files follow. Some commands used in these files are not covered until later in this chapter. In any startup file, you must place in the environment (export) those variables and functions that you want to be available to child processes. For more information refer to "Environment, Environment Variables, and Inheritance" on page 480.

```
$ cat ~/.bash_profile
if [ -f ~/.bashrc ]; then
    . ~/.bashrc                # Read local startup file if it exists
fi
PATH=$PATH:/usr/local/bin      # Add /usr/local/bin to PATH
export PS1='[\h \W \!]\$ '     # Set prompt
```

The first command in the preceding **.bash_profile** file executes the commands in the user's **.bashrc** file if it exists. The next command adds to the **PATH** variable (page 318). Typically, **PATH** is set and exported in **/etc/profile**, so it does not need to be exported in a user's startup file. The final command sets and exports **PS1** (page 319), which controls the user's prompt.

The first command in the **.bashrc** file shown below executes the commands in the **/etc/bashrc** file if it exists. Next, the file sets **noclobber** (page 143), unsets **MAILCHECK** (page 319), exports **LANG** (page 324) and **VIMINIT** (for vim initialization; page 202), and defines several aliases. The final command defines a function (page 356) that swaps the names of two files.

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

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

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

. (Dot) or source: Runs a Startup File in the Current Shell

After you edit a startup file such as **.bashrc**, you do not have to log out and log in again to put the changes into effect. Instead, you can run the startup file using the

. (dot) or source builtin (they are the same command under bash; only source is available under tcsh [page 421]). As with other commands, the . must be followed by a SPACE on the command line. Using . or source is similar to running a shell script, except these commands run the script as part of the current process. Consequently, when you use . or source to run a script, changes you make to variables from within the script affect the shell you run the script from. If you ran a startup file as a regular shell script and did not use the . or source builtin, the variables created in the startup file would remain in effect only in the subshell running the script—not in the shell you ran the script from. You can use the . or source command to run any shell script—not just a startup file—but undesirable side effects (such as changes in the values of shell variables you rely on) might occur. For more information refer to "Environment, Environment Variables, and Inheritance" on page 480.

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

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

$ . ~/.bashrc
guava:
```

COMMANDS THAT ARE SYMBOLS

The Bourne Again Shell uses the symbols (,), [,], and $ in a variety of ways. To minimize confusion, Table 8-1 lists the most common use of each of these symbols and the page on which it is discussed.

Table 8-1 Builtin commands that are symbols

Symbol	Command
( )	Subshell (page 302)
$( )	Command substitution (page 371)
(( ))	Arithmetic evaluation; a synonym for let (use when the enclosed value contains an equal sign; page 505)
$(( ))	Arithmetic expansion (not for use with an enclosed equal sign; page 369)
[ ]	The test command (pages 431, 434, and 1005)
[[ ]]	Conditional expression; similar to [ ] but adds string comparisons (page 506)

Redirecting Standard Error

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

Just as it does with standard output, by default the shell directs standard error to the screen. Unless you redirect one or the other, you might not know the difference between the output a command sends to standard output and the output it sends to standard error. One difference is that the system buffers standard output but does not buffer standard error. This section describes the syntax used by bash to redirect standard error and to distinguish between standard output and standard error. See page 389 if you are using tcsh.

File descriptors

A *file descriptor* is the place a program sends its output to and gets its input from. When you execute a program, the shell opens three file descriptors for the program: 0 (standard input), 1 (standard output), and 2 (standard error). The redirect output symbol (**>** [page 140]) is shorthand for **1>**, which tells the shell to redirect standard output. Similarly **<** (page 142) is short for **0<**, which redirects standard input. The symbols **2>** redirect standard error. For more information refer to "File Descriptors" on page 464.

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

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

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

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

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

Similarly, when you send standard output through a pipeline, standard error is not affected. The following example sends standard output of cat through a pipeline to tr (page 1014), which in this example converts lowercase characters to uppercase.

The text that cat sends to standard error is not translated because it goes directly to the screen rather than through the pipeline.

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

The following example redirects standard output and standard error to different files. The shell redirects standard output (file descriptor 1) to the filename following **1>**. You can specify **>** in place of **1>**. The shell redirects standard error (file descriptor 2) to the filename following **2>**.

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

Combining standard output and standard error

In the next example, the **&>** token redirects standard output and standard error to a single file. The **>&** token performs the same function under tcsh (page 389).

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

Duplicating a file descriptor

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

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

In this case, **1> hold** precedes **2>&1**. If they had appeared in the opposite order, standard error would have been made a duplicate of standard output before standard output was redirected to **hold**. Only standard output would have been redirected to **hold** in that case.

Sending errors through a pipeline

The next example declares file descriptor 2 to be a duplicate of file descriptor 1 and sends the output for file descriptor 1 (as well as file descriptor 2) through a pipeline to the tr command.

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

The token **|&** is shorthand for **2>&1 |**:

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

Sending errors to standard error

You can use **1>&2** (or simply **>&2**; the **1** is not required) to redirect standard output of a command to standard error. Shell scripts use this technique to send the output of echo to standard error. In the following script, standard output of the first echo is redirected to standard error:

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

If you redirect standard output of **message_demo**, error messages such as the one produced by the first echo appear on the screen because you have not redirected standard error. Because standard output of a shell script is frequently redirected to a file, you can use this technique to display on the screen any error messages generated by the script. The **lnks** script (page 439) uses this technique. You can use the exec builtin to create additional file descriptors and to redirect standard input, standard output, and standard error of a shell script from within the script (page 494).

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

Table 8-2 Redirection operators

Operator	Meaning
< ***filename***	Redirects standard input from ***filename.***
> ***filename***	Redirects standard output to ***filename*** unless ***filename*** exists and **noclobber** (page 143) is set. If **noclobber** is not set, this redirection creates ***filename*** if it does not exist and overwrites it if it does exist.
>! ***filename***	Redirects standard output to ***filename,*** even if the file exists and **noclobber** (page 143) is set.
>> ***filename***	Redirects and appends standard output to ***filename;*** creates ***filename*** if it does not exist.
&> ***filename***	Redirects standard output and standard error to ***filename***.
<&***m***	Duplicates standard input from file descriptor ***m*** (page 465).
[n]>&***m***	Duplicates standard output or file descriptor ***n*** if specified from file descriptor ***m*** (page 465).
[n]<&–	Closes standard input or file descriptor ***n*** if specified (page 465).
[n]>&–	Closes standard output or file descriptor ***n*** if specified.

Writing and Executing a Simple Shell Script

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

In addition to the commands you would ordinarily use on the command line, *control flow* commands (also called *control structures*) find most of their use in shell scripts. This group of commands enables you to alter the order of execution of commands in a script in the same way you would alter the order of execution of statements using a structured programming language. Refer to “Control Structures” on page 430 (bash) and page 408 (tcsh) for specifics.

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

chmod: Makes a File Executable

To execute a shell script by giving its name as a command, you must have permission to read and execute the file that contains the script (refer to “Access Permissions” on page 100). Read permission enables you to read the file that holds the script. Execute permission tells the system that the owner, group, and/or public has permission to execute the file; it implies the content of the file is executable.

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

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

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

You cannot execute **whoson** by giving its name as a command because you do not have execute permission for the file. The system does not recognize **whoson** as an executable file and issues the error message **Permission denied** when you try to execute it. (See the tip on the next page if the shell issues a **command not found** error message.) When you give the filename as an argument to bash (**bash whoson**), bash assumes the argument is a shell script and executes it. In this case **bash** is executable, and **whoson** is an argument that bash executes, so you do not need execute permission to **whoson**. You must have read permission.

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

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

Command not found?

tip If you give the name of a shell script as a command without including the leading **./**, the shell typically displays the following error message:

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

This message indicates the shell is not set up to search for executable files in the working directory. Enter this command instead:

```
$ ./whoson
```

The **./** tells the shell explicitly to look for an executable file in the working directory. Although not recommended for security reasons, you can change the **PATH** variable so the shell searches the working directory automatically; see **PATH** on page 318.

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

The final command in Figure 8-1 shows the shell executing the file when its name is given as a command. For more information refer to "Access Permissions" on page 100 as well as the discussions of ls and chmod in Part VII.

```
$ ls -l whoson
-rw-rw-r--. 1 max pubs 40 05-24 11:30 whoson

$ chmod u+x whoson
$ ls -l whoson
-rwxrw-r--. 1 max pubs 40 05-24 11:30 whoson

$ ./whoson
Fri May 25 11:40:49 PDT 2018
Users Currently Logged In
zach     pts/7    2018-05-23 18:17
hls      pts/1    2018-05-24 09:59
sam      pts/12   2018-05-24 06:29 (guava)
max      pts/4    2018-05-24 09:08
```

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

#! SPECIFIES A SHELL

You can put a special sequence of characters on the first line of a shell script to tell the operating system which shell (or other program) should execute the file and which options you want to include. Because the operating system checks the initial characters of a program before attempting to execute it using **exec**, these characters save the system from making an unsuccessful attempt. If **#!** (sometimes said out loud as *hashbang* or *shebang*) are the first two characters of a script, the system interprets the characters that follow as the absolute pathname of the program that is to execute the script. This pathname can point to any program, not just a shell, and can be useful if you have a script you want to run with a shell other than the shell you are running the script from. The following example specifies that bash should run the script:

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

The bash –e and –u options can make your programs less fractious

tip The bash **–e** (errexit) option causes bash to exit when a simple command (e.g., not a control structure) fails. The bash **–u** (nounset) option causes bash to display a message and exit when it tries to expand an unset variable. See Table 8-13 on page 361 for details. It is easy to turn these options on in the **!#** line of a bash script:

```
#!/bin/bash -eu
```

These options can prevent disaster when you mistype lines like this in a script:

```
MYDIR=/tmp/$$
cd $MYDIr; rm -rf .
```

During development, you can also specify the **–x** option in the **!#** line to turn on debugging (page 442).

The next example runs under Perl and can be run directly from the shell without explicitly calling Perl on the command line:

```
$ cat ./perl_script.pl
#!/usr/bin/perl -w
print "This is a Perl script.\n";

$ ./perl_script.pl
This is a Perl script.
```

The next example shows a script that should be executed by tcsh:

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

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

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

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

$ ./tcsh_script2
UID        PID  PPID  C STIME TTY          TIME CMD
max       3031  3030  0 Nov16 pts/4    00:00:00 -bash
max       9358  3031  0 21:13 pts/4    00:00:00 /bin/tcsh ./tcsh_script2
max       9375  9358  0 21:13 pts/4    00:00:00 ps -f
```

If you do not follow **#!** with the name of an executable program, the shell reports it cannot find the program you asked it to run. You can optionally follow **#!** with SPACEs before the name of the program. If you omit the **#!** line and try to run, for example, a tcsh script from bash, the script will run under bash and might generate error messages or not run properly. See page 682 for an example of a stand-alone sed script that uses **#!**.

Begins a Comment

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

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

Executing a Shell Script

fork and **exec** system calls

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

```
$ bash whoson
```

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

```
$ bash < whoson
```

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

sh does not call the original Bourne Shell

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

Control Operators: Separate and Group Commands

Whether you give the shell commands interactively or write a shell script, you must separate commands from one another. This section, which applies to the Bourne Again and TC Shells, reviews the ways to separate commands that were covered in Chapter 5 and introduces a few new ones.

The tokens that separate, terminate, and group commands are called *control operators*. Each of the control operators implies line continuation as explained on page 512. Following is a list of the control operators and the page each is discussed on.

- ; Command separator (next page)
- NEWLINE Command initiator (next page)
- & Background task (next page)
- | Pipeline (next page)
- |& Standard error pipeline (page 293)
- () Groups commands (page 302)
- || Boolean OR (page 302)
- && Boolean AND (page 302)
- ;; Case terminator (page 454)

; AND NEWLINE SEPARATE COMMANDS

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

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

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

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

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

Whitespace Although the whitespace (SPACEs and/or TABs) around the semicolons in the previous example makes the command line easier to read, it is not necessary. None of the control operators needs to be surrounded by whitespace.

| AND & SEPARATE COMMANDS AND DO SOMETHING ELSE

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

Each of the following command lines initiates a pipeline (page 145) comprising three simple commands:

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

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

The next command line executes a list (page 149) by running the simple commands **d** and **e** in the background and the simple command **f** in the foreground:

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

The shell displays the job number between brackets and the PID number for each process running in the background. It displays a prompt as soon as **f** finishes, which might be before **d** or **e** finishes.

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

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

The next command line executes a list that runs three commands as background jobs. The shell displays a shell prompt immediately:

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

The next example uses a pipe symbol to send the output from one command to the next command and an ampersand (&) to run the entire pipeline in the background. Again, the shell displays the prompt immediately. The shell commands that are part of a pipeline form a single job. That is, the shell treats a pipeline as a single job, no matter how many commands are connected using pipe (**|**) symbols or how complex they are. The Bourne Again Shell reports only one process in the background (although there are three):

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

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

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

&& and || Boolean Control Operators

The && (AND) and **||** (OR) Boolean operators are called *short-circuiting* control operators. If the result of using one of these operators can be decided by looking only at the left operand, the right operand is not evaluated. The result of a Boolean operation is either 0 (*true*) or 1 (*false*).

&& The && operator causes the shell to test the exit status of the command preceding it. If the command succeeds, bash executes the next command; otherwise, it skips the next command. You can use this construct to execute commands conditionally.

```
$ mkdir bkup && cp -r src bkup
```

This compound command creates the directory **bkup**. If mkdir succeeds, the content of directory **src** is copied recursively to **bkup**.

|| The || control operator also causes bash to test the exit status of the first command but has the opposite effect: The remaining command(s) are executed only if the first command failed (that is, exited with nonzero status).

```
$ mkdir bkup || echo "mkdir of bkup failed" >> /tmp/log
```

The exit status of a command list is the exit status of the last command in the list. You can group lists with parentheses. For example, you could combine the previous two examples as

```
$ (mkdir bkup && cp -r src bkup) || echo "mkdir failed" >> /tmp/log
```

In the absence of parentheses, && and || have equal precedence and are grouped from left to right. The following examples use the true and false utilities. These utilities do nothing and return *true* (0) and *false* (1) exit statuses, respectively:

```
$ false; echo $?
1
```

The **$?** variable holds the exit status of the preceding command (page 477). The next two commands yield an exit status of 1 (*false*):

```
$ true || false && false
$ echo $?
1
$ (true || false) && false
$ echo $?
1
```

Similarly, the next two commands yield an exit status of 0 (*true*):

```
$ false && false || true
$ echo $?
0
$ (false && false) || true
$ echo $?
0
```

See "Lists" on page 149 for more examples.

optional

( ) Groups Commands

You can use the parentheses control operator to group commands. When you use this technique, the shell creates a copy of itself, called a *subshell,* for each group. It treats each group of commands as a list and creates a new process to execute each command (refer to "Process Structure" on page 333 for more information on creating subshells). Each subshell has its own environment, meaning it has its own set of variables whose values can differ from those in other subshells.

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

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

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

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

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

The next script copies one directory to another. The second pair of parentheses creates a subshell to run the commands following the pipe symbol. Because of these parentheses, the output of the first tar command is available for the second tar command, despite the intervening cd command. Without the parentheses, the output of the first tar command would be sent to cd and lost because cd does not process standard input. The shell variables **$1** and **$2** hold the first and second command-line arguments (page 471), respectively. The first pair of parentheses, which creates a subshell to run the first two commands, allows users to call **cpdir** with relative pathnames. Without them, the first cd command would change the working directory of the script (and consequently the working directory of the second cd command). With them, only the working directory of the subshell is changed.

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

The **cpdir** command line copies the files and directories in the **/home/max/sources** directory to the directory named **/home/max/memo/biblio**. Running this shell script is the same as using cp with the **–r** option. See page 772 for more information on cp.

\ Continues a Command

Although it is not a control operator, you can use a backslash (\) character in the middle of commands. When you enter a long command line and the cursor reaches the right side of the screen, you can use a backslash to continue the command on the next line. The backslash quotes, or escapes, the NEWLINE character that follows it so the shell does not treat the NEWLINE as a control operator. Enclosing a backslash within single quotation marks or preceding it with another backslash turns off the power of a backslash to quote special characters such as NEWLINE (not tcsh; see **prompt2** on page 404). Enclosing a backslash within double quotation marks has no effect on the power of the backslash (not tcsh).

Although you can break a line in the middle of a word (token), it is typically simpler, and makes code easier to read, if you break a line immediately before or after whitespace.

optional You can enter a RETURN in the middle of a quoted string on a command line without using a backslash. (See **prompt2** on page 404 for tcsh behavior.) The NEWLINE (RETURN) you enter will then be part of the string:

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

In the three examples in this section, the shell does not interpret RETURN as a control operator because it occurs within a quoted string. The greater than sign (**>**) is a secondary prompt (**PS2**; page 321) indicating the shell is waiting for you to continue the unfinished command. In the next example, the first RETURN is quoted (escaped) so the shell treats it as a separator and does not interpret it literally.

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

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

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

Job Control

As explained on page 150, a *job* is another name for a process running a pipeline (which can be a simple command). You run one or more jobs whenever you give the shell a command. For example, if you type **date** on the command line and press RETURN, you have run a job. You can also create several jobs on a single command line by entering several simple commands separated by control operators (& in the following example):

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

The portion of the command line up to the first & is one job—a pipeline comprising three simple commands connected by pipe symbols: find, sort, and lpr. The second job is a pipeline that is a simple command (grep). The & characters following each pipeline put each job in the background, so bash does not wait for them to complete before displaying a prompt.

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

jobs: Lists Jobs

The jobs builtin lists all background jobs. In the following example, the sleep command runs in the background and creates a background job that jobs reports on:

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

fg: Brings a Job to the Foreground

The shell assigns a job number to each job you run in the background. For each job run in the background, the shell lists the job number and PID number immediately, just before it issues a prompt:

```
$ gnome-calculator &
[1] 1246
$ date &
[2] 1247
$ Fri Dec 7 11:44:40 PST 2018
[2]+ Done          date
$ find /usr -name ace -print > findout &
[2] 1269
$ jobs
[1]- Running        gnome-calculator &
[2]+ Running        find /usr -name ace -print > findout &
```

The shell discards job numbers when a job is finished and reuses discarded job numbers. When you start or put a job in the background, the shell assigns a job number that is one more than the highest job number in use.

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

To move a background job to the foreground, use the fg builtin followed by the job number. Alternatively, you can give a percent sign (%) followed by the job number as a command. Either of the following commands moves job 2 to the foreground.

When you move a job to the foreground, the shell displays the command it is now executing in the foreground.

```
$ fg 2
find /usr -name ace -print > findout
```

or

```
$ %2
find /usr -name ace -print > findout
```

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

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

SUSPENDING A JOB

Pressing the suspend key (usually CONTROL-Z) immediately suspends (temporarily stops) the job in the foreground and displays a message that includes the word **Stopped**.

```
CONTROL-Z
[2]+  Stopped                 find /usr -name ace -print > findout
```

For more information refer to "Moving a Job from the Foreground to the Background" on page 151.

bg: SENDS A JOB TO THE BACKGROUND

To move the foreground job to the background, you must first suspend the job (above). You can then use the bg builtin to resume execution of the job in the background.

```
$ bg
[2]+ find /usr -name ace -print > findout &
```

If a background job attempts to read from the terminal, the shell stops the job and displays a message saying the job has been stopped. You must then move the job to the foreground so it can read from the terminal.

```
$ (sleep 5; cat > mytext) &
[1] 1343
$ date
Fri Dec  7 11:58:20 PST 2018
```

```
[1]+ Stopped                  ( sleep 5; cat >mytext )
$ fg
( sleep 5; cat >mytext )
Remember to let the cat out!
CONTROL-D
$
```

In the preceding example, the shell displays the job number and PID number of the background job as soon as it starts, followed by a prompt. Demonstrating that you can give a command at this point, the user gives the command date, and its output appears on the screen. The shell waits until just before it issues a prompt (after date has finished) to notify you that job 1 is stopped. When you give an fg command, the shell puts the job in the foreground, and you can enter the data the command is waiting for. In this case the input needs to be terminated using CONTROL-D, which sends an EOF (end of file) signal to cat. The shell then displays another prompt.

The shell keeps you informed about changes in the status of a job, notifying you when a background job starts, completes, or stops, perhaps because it is waiting for input from the terminal. The shell also lets you know when a foreground job is suspended. Because notices about a job being run in the background can disrupt your work, the shell delays displaying these notices until just before it displays a prompt. You can set **notify** (page 363) to cause the shell to display these notices without delay.

If you try to exit from a nonlogin shell while jobs are stopped, the shell issues a warning and does not allow you to exit. If you then use jobs to review the list of jobs or you immediately try to exit from the shell again, the shell allows you to exit. If **huponexit** (page 362) is not set (it is not set by default), stopped jobs remain stopped and background jobs keep running in the background. If it is set, the shell terminates these jobs.

MANIPULATING THE DIRECTORY STACK

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

dirs: DISPLAYS THE STACK

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

```
$ dirs
~/literature
```

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

pushd: Pushes a Directory on the Stack

When you supply the pushd (push directory) builtin with one argument, it pushes the directory specified by the argument on the stack, changes directories to the specified directory, and displays the stack. The following example is illustrated in Figure 8-3:

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

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

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

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

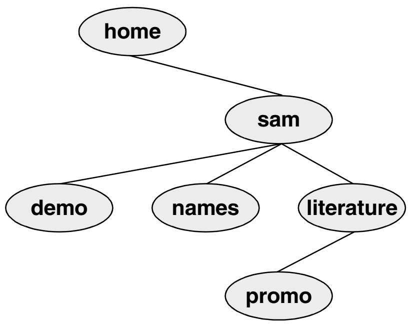

Figure 8-2 The directory structure used in the examples

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

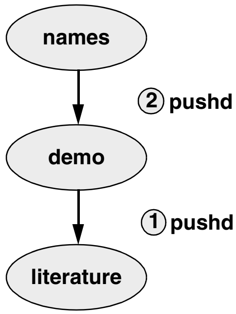

Figure 8-3 Creating a directory stack

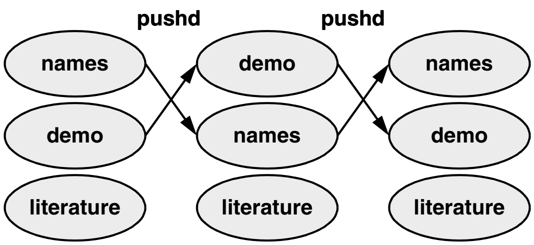

Figure 8-4 Using pushd to change working directories

popd: Pops a Directory Off the Stack

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

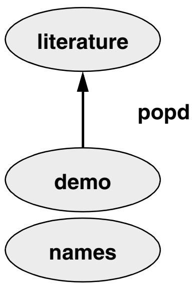

Figure 8-5 Using popd to remove a directory from the stack

top directory from the stack and changes the working directory to the new top directory:

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

To remove a directory other than the top one from the stack, use popd with a numeric argument preceded by a plus sign. The following example removes directory number 1, **demo**. Removing a directory other than directory number 0 does not change the working directory.

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

Parameters and Variables

Shell parameter: Within a shell, a *shell parameter* is associated with a value you or a shell script can access. This section introduces the following kinds of shell parameters: user-created variables, keyword variables, positional parameters, and special parameters.

Variables: Parameters whose names consist of letters, digits, and underscores are referred to as *variables*. A variable name must start with a letter or underscore, not with a number. Thus, **A76**, **MY_CAT**, and **___X___** are valid variable names, whereas **69TH_STREET** (starts with a digit) and **MY-NAME** (contains a hyphen) are not.

User-created variables: Variables that you name and assign values to are *user-created variables*. You can change the values of user-created variables at any time, or you can make them *readonly* so that their values cannot be changed.

Shell variables and environment variables: By default, a variable is available only in the shell it was created in (i.e., local); this type of variable is called a *shell variable*. You can use export to make a variable available in shells spawned from the shell it was created in (i.e., global); this type of variable is called an *environment variable*. One naming convention is to use mixed-case or lowercase letters for shell variables and only uppercase letters for environment variables. Refer to "Variables" on page 479 for more information on shell variables and environment variables.

To declare and initialize a variable in bash, use the following syntax:

VARIABLE=value

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

```
$ myvar=abc
```

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

```
$ set myvar = abc
```

Declaring and initializing a variable for a script

The Bourne Again Shell permits you to put variable assignments at the beginning of a command line. This type of assignment places variables in the environment of the command shell—that is, the variable is accessible only from the program (and the children of the program) the command runs. It is not available from the shell running the command. The **my_script** shell script displays the value of **TEMPDIR**. The following command runs **my_script** with **TEMPDIR** set to **/home/sam/temp**. The echo builtin shows that the interactive shell has no value for **TEMPDIR** after running **my_script**. If **TEMPDIR** had been set in the interactive shell, running **my_script** in this manner would have had no effect on its value.

```
$ cat my_script
echo $TEMPDIR
$ TEMPDIR=/home/sam/temp ./my_script
/home/sam/temp
$ echo $TEMPDIR

$
```

Keyword variables

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

You can change the values of most keyword shell variables. It is usually not necessary to change the values of keyword variables initialized in the **/etc/profile** or **/etc/csh.cshrc** systemwide startup files. If you need to change the value of a bash keyword variable, do so in one of your startup files (for bash see page 288; for tcsh see page 382). Just as you can make user-created variables environment variables, so you can make keyword variables environment variables—a task usually done automatically in startup files. You can also make a keyword variable readonly. See page 317 for a discussion of keyword variables.

Positional and special parameters

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

Whenever you run a command, each argument on the command line becomes the value of a *positional parameter* (page 470). Positional parameters enable you to access command-line arguments, a capability you will often require when you write shell scripts. The set builtin (page 472) enables you to assign values to positional parameters.

Other frequently needed shell script values, such as the name of the last command executed, the number of positional parameters, and the status of the most recently executed command, are available as *special parameters* (page 475). You cannot assign values to special parameters.

User-Created Variables

The first line in the following example declares the variable named **person** and initializes it with the value **max**:

```
$ person=max
$ echo person
person
$ echo $person
max
```

Parameter substitution

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

Quoting the $

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

```
$ echo $person
max
$ echo "$person"
max
$ echo '$person'
$person
$ echo \$person
$person
```

SPACES

Because they do not prevent variable substitution but do turn off the special meanings of most other characters, double quotation marks are useful when you assign values to variables and when you use those values. To assign a value that contains SPACEs or

TABs to a variable, use double quotation marks around the value. Although double quotation marks are not required in all cases, using them is a good habit.

```
$ person="max and zach"
$ echo $person
max and zach
$ person=max and zach
bash: and: command not found
```

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

```
$ person="max   and   zach"
$ echo $person
max and zach
$ echo "$person"
max   and   zach
```

Pathname expansion in assignments

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

The first line in the following sequence of commands assigns the string **max*** to the variable **memo**. All shells interpret special characters as special when you reference a variable that contains an unquoted special character. In the following example, the shell expands the value of the **memo** variable because it is not quoted:

```
$ memo=max*
$ ls
max.report
max.summary
$ echo $memo
max.report max.summary
```

Above, the shell expands the **$memo** variable to **max***, expands **max*** to **max.report** and **max.summary**, and passes these two values to echo. In the next example, the Bourne Again Shell *does not expand the string* because bash does not perform pathname expansion (page 152) when it assigns a value to a variable.

```
$ echo "$memo"
max*
```

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

optional

Braces around variables

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

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

$
```

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

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

The Bourne Again Shell refers to command-line arguments using the positional parameters **$1**, **$2**, **$3**, and so forth up to **$9**. You must use braces to refer to arguments past the ninth argument: **${10}**. The name of the command is held in **$0** (page 470).

unset: Removes a Variable

Unless you remove a variable, it exists as long as the shell in which it was created exists. To remove the *value* of a variable but not the variable itself, assign a null value to the variable. In the following example, set (page 472) displays a list of all variables and their values; grep extracts the line that shows the value of **person**.

```
$ echo $person
zach
$ person=
$ echo $person

$ set | grep person
person=
```

You can remove a variable using the unset builtin. The following command removes the variable **person**:

```
$ unset person
$ echo $person

$ set | grep person
$
```

Variable Attributes

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

readonly: Makes the Value of a Variable Permanent

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

```
$ person=zach
$ echo $person
zach
$ readonly person
$ person=helen
bash: person: readonly variable
$ unset person
bash: unset: person: cannot unset: readonly variable
```

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

declare: Lists and Assigns Attributes to Variables

The declare builtin (not in tcsh) lists and sets attributes and values for shell variables. The typeset builtin (another name for declare) performs the same function but is deprecated. Table 8-3 lists five of these attributes.

Table 8-3 Variable attributes (declare)

Attribute	Meaning
–a	Declares a variable as an array (page 486)
–f	Declares a variable to be a function name (page 356)
–i	Declares a variable to be of type integer (page 316)
–r	Makes a variable readonly; also readonly (above)
–x	Makes a variable an environment variable; also export (page 480)

The following commands declare several variables and set some attributes. The first line declares **person1** and initializes it to **max**. This command has the same effect with or without the word **declare**.

```
$ declare person1=max
$ declare -r person2=zach
$ declare -rx person3=helen
$ declare -x person4
```

readonly and export The **readonly** and **export** builtins are synonyms for the commands **declare –r** and **declare –x**, respectively. You can declare a variable without initializing it, as the preceding declaration of the variable **person4** illustrates. This declaration makes **person4** an environment variable so it is available to all subshells. Until **person4** is initialized, it has a null value.

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

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

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

```
$ declare +x person3
```

See page 481 for more information on exporting variables.

Listing variable attributes Without any arguments or options, declare lists all shell variables. The same list is output when you run set (page 473) without any arguments.

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

```
$ declare -r
declare -r BASHOPTS="checkwinsize:cmdhist:expand_aliases: ... "
declare -ir BASHPID
declare -ar BASH_VERSINFO='([0]="4" [1]="2" [2]="24" [3]="1" ... '
declare -ir EUID="500"
declare -ir PPID="1936"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand: ... "
declare -ir UID="500"
declare -r person2="zach"
declare -rx person3="helen"
```

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

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

```
$ declare -i COUNT
```

You can use declare to display integer variables:

```
$ declare -i
declare -ir BASHPID
declare -i COUNT
declare -ir EUID="1000"
declare -i HISTCMD
declare -i LINENO
declare -i MAILCHECK="60"
declare -i OPTIND="1"
...
```

KEYWORD VARIABLES

Keyword variables are either inherited or declared and initialized by the shell when it starts. You can assign values to these variables from the command line or from a startup file. Typically, these variables are environment variables (exported) so they are available to subshells you start as well as your login shell.

HOME: YOUR HOME DIRECTORY

By default, your home directory is the working directory when you log in. Your home directory is established when your account is set up; under Linux its name is stored in the **/etc/passwd** file. macOS uses Open Directory (page 1068) to store this information.

```
$ grep sam /etc/passwd
sam:x:500:500:Sam the Great:/home/sam:/bin/bash
```

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

```
$ pwd
/home/max/laptop
$ echo $HOME
/home/max
$ cd
$ pwd
/home/max
```

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

Tilde (~) The shell uses the value of **HOME** to expand pathnames that use the shorthand tilde (~) notation (page 91) to denote a user's home directory. The following example uses echo to display the value of this shortcut and then uses ls to list the files in Max's **laptop** directory, which is a subdirectory of his home directory:

```
$ echo ~
/home/max
$ ls ~/laptop
tester     count      lineup
```

PATH: Where the Shell Looks for Programs

When you give the shell an absolute or relative pathname as a command, it looks in the specified directory for an executable file with the specified filename. If the file with the pathname you specified does not exist, the shell reports **No such file or directory**. If the file exists as specified but you do not have execute permission for it, or in the case of a shell script you do not have read and execute permission for it, the shell reports **Permission denied**.

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

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

Working directory

The **PATH** variable specifies the directories in the order the shell should search them. Each directory must be separated from the next by a colon. The following command sets **PATH** so a search for an executable file starts with the **/usr/local/bin** directory. If it does not find the file in this directory, the shell looks next in **/bin** and then in **/usr/bin**. If the search fails in those directories, the shell looks in the **~/bin** directory, a subdirectory of the user's home directory. Finally, the shell looks in the working directory. Exporting **PATH** makes sure it is an environment variable so it is available to subshells, although it is typically exported when it is declared so exporting it again is not necessary:

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

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

Because Linux stores many executable files in directories named **bin** (*binary*), users typically put their executable files in their own **~/bin** directories. If you put your own **bin** directory toward the end of **PATH**, as in the preceding example, the shell looks there for any commands it cannot find in directories listed earlier in **PATH**.

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

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

Set PATH in **~/.bash_profile**; see the tip on page 289.

PATH and security

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

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

MAIL: Where Your Mail Is Kept

The **MAIL** variable (**mail** under tcsh) usually contains the pathname of the file that holds your mail (your *mailbox,* usually **/var/mail/*name*,** where ***name*** is your username). However, you can use **MAIL** to watch any file (including a directory): Set **MAIL** to the name of the file you want to watch.

If **MAIL** is set and **MAILPATH** (below) is not set, the shell informs you when the file specified by **MAIL** is modified (such as when mail arrives). In a graphical environment you can unset **MAIL** so the shell does not display mail reminders in a terminal emulator window (assuming you are using a graphical mail program).

Most macOS systems do not use local files for incoming mail. Instead, mail is typically kept on a remote mail server. The **MAIL** variable and other mail-related shell variables have no effect unless you have a local mail server.

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

The **MAILCHECK** variable (not in tcsh) specifies how often, in seconds, the shell checks the directories specified by **MAIL** or **MAILPATH**. The default is 60 seconds. If you set this variable to zero, the shell checks before it issues each prompt.

PS1: User Prompt (Primary)

The default Bourne Again Shell prompt is a dollar sign (**$**). When you run bash with **root** privileges, bash typically displays a hashmark (**#**) prompt. The **PS1** variable (**prompt** under tcsh; page 403) holds the prompt string the shell uses to let you know it is waiting for a command. When you change the value of **PS1**, you change the appearance of your prompt.

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

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

displays the prompt

[user@host directory event]$

where ***user*** is the username, ***host*** is the hostname up to the first period, ***directory*** is the basename of the working directory, and ***event*** is the event number (page 337) of the current command.

If you are working on more than one system, it can be helpful to incorporate the system name into your prompt. The first example that follows changes the prompt to the name of the local host, a SPACE, and a dollar sign (or, if the user is running with **root** privileges, a hashmark), followed by a SPACE. A SPACE at the end of the prompt makes commands you enter following the prompt easier to read. The second example changes the prompt to the time followed by the name of the user. The third example changes the prompt to the one used in this book (a hashmark for a user running with **root** privileges and a dollar sign otherwise):

```
$ PS1='\h \$ '
guava $

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

$ PS1='\$ '
$
```

Table 8-4 describes some of the symbols you can use in **PS1**. See Table 9-4 on page 403 for the corresponding tcsh symbols. For a complete list of special characters you can use in the prompt strings, open the bash man page and search for the third occurrence of **PROMPTING** (enter the command **/PROMPTING** followed by a RETURN and then press **n** two times).

Table 8-4 **PS1** symbols

Symbol	Display in prompt
\$	# if the user is running with **root** privileges; otherwise, **$**
\w	Pathname of the working directory
\W	Basename of the working directory
\!	Current event (history) number (page 341)
\d	Date in Weekday Month Date format
\h	Machine hostname, without the domain
\H	Full machine hostname, including the domain
\u	Username of the current user

Table 8-4 **PS1** symbols (continued)

Symbol	Display in prompt
\@	Current time of day in 12-hour, AM/PM format
\T	Current time of day in 12-hour HH:MM:SS format
\A	Current time of day in 24-hour HH:MM format
\t	Current time of day in 24-hour HH:MM:SS format

PS2: User Prompt (Secondary)

The **PS2** variable holds the secondary prompt (**prompt2** under tcsh). On the first line of the next example, an unclosed quoted string follows echo. The shell assumes the command is not finished and on the second line displays the default secondary prompt (>). This prompt indicates the shell is waiting for the user to continue the command line. The shell waits until it receives the quotation mark that closes the string and then executes the command:

```
$ echo "demonstration of prompt string
> 2"
demonstration of prompt string
2
```

The next command changes the secondary prompt to **Input =>** followed by a SPACE. On the line with who, a pipe symbol (|) implies the command line is continued (page 512) and causes bash to display the new secondary prompt. The command **grep sam** (followed by a RETURN) completes the command; grep displays its output.

```
$ PS2="Input => "
$ who |
Input => grep sam
sam      tty1         2018-05-01 10:37 (:0)
```

PS3: Menu Prompt

The **PS3** variable holds the menu prompt (**prompt3** in tcsh) for the **select** control structure (page 461).

PS4: Debugging Prompt

The **PS4** variable holds the bash debugging symbol (page 443; not in tcsh).

IFS: Separates Input Fields (Word Splitting)

The **IFS** (Internal Field Separator) shell variable (not in tcsh) specifies the characters you can use to separate arguments on a command line. It has the default value of SPACE-TAB-NEWLINE. Regardless of the value of **IFS**, you can always use one or more SPACE or TAB characters to separate arguments on the command line, provided these characters are not quoted or escaped. When you assign character values to **IFS**, these characters can also separate fields—but only if they undergo expansion. This type of interpretation of the command line is called *word splitting* and is discussed on page 372.

Be careful when changing IFS

caution Changing **IFS** has a variety of side effects, so work cautiously. You might find it useful to save the value of **IFS** before changing it. You can then easily restore the original value if a change yields unexpected results. Alternatively, you can fork a new shell using a **bash** command before experimenting with **IFS**; if you run into trouble, you can **exit** back to the old shell, where **IFS** is working properly.

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

```
$ a=w:x:y:z

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

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

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

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

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

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

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

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

This time expansion occurs, so the **p** in the token **export** is interpreted as a separator (as the echo command shows). Next, when you try to use the value of the **aa** variable to export the **VAR** variable, the shell parses the **$aa VAR** command line as **ex ort VAR**. The effect is that the command line starts the ex editor with two filenames: **ort** and **VAR**.

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

If **IFS** is unset, bash uses its default value (SPACE-TAB-NEWLINE). If **IFS** is null, bash does not split words.

Multiple separator characters

tip Although the shell treats sequences of multiple SPACE or TAB characters as a single separator, it treats *each occurrence* of another field-separator character as a separator.

CDPATH: Broadens the Scope of cd

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

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

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

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

This command causes cd to search your home directory, the **literature** directory, and then the working directory when you give a cd command. If you do not include the working directory in **CDPATH**, cd searches the working directory if the search of all the other directories in **CDPATH** fails. If you want cd to search the working directory first, include a colon (**:**) as the first entry in **CDPATH**:

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

If the argument to the cd builtin is anything other than a simple filename (i.e., if the argument contains a slash [**/**]), the shell does not consult **CDPATH**.

Keyword Variables: A Summary

Table 8-5 lists the bash keyword variables. See page 402 for information on tcsh variables.

Table 8-5 bash keyword variables

Variable	Value
BASH_ENV	The pathname of the startup file for noninteractive shells (page 289)
CDPATH	The cd search path (page 323)
COLUMNS	The width of the display used by **select** (page 460)
HISTFILE	The pathname of the file that holds the history list (default: **~/.bash_history**; page 336)
HISTFILESIZE	The maximum number of entries saved in **HISTFILE** (default: 1,000–2,000; page 336)
HISTSIZE	The maximum number of entries saved in the history list (default: 1,000; page 336)
HOME	The pathname of the user's home directory (page 317); used as the default argument for cd and in tilde expansion (page 91)
IFS	Internal Field Separator (page 321); used for word splitting (page 372)
INPUTRC	The pathname of the Readline startup file (default: **~/.inputrc**; page 349)
LANG	The locale category when that category is not specifically set using one of the **LC_** variables (page 327)
LC_	A group of variables that specify locale categories including **LC_ALL**, **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, and **LC_NUMERIC**; use the locale builtin (page 328) to display a more complete list including values
LINES	The height of the display used by **select** (page 460)
MAIL	The pathname of the file that holds a user's mail (page 319)
MAILCHECK	How often, in seconds, bash checks for mail (default: 60; page 319)
MAILPATH	A colon-separated list of file pathnames that bash checks for mail in (page 319)
OLDPWD	The pathname of the previous working directory
PATH	A colon-separated list of directory pathnames that bash looks for commands in (page 318)
PROMPT_COMMAND	A command that bash executes just before it displays the primary prompt

Table 8-5 bash keyword variables (continued)

Variable	Value
PS1	Prompt String 1; the primary prompt (page 319)
PS2	Prompt String 2; the secondary prompt (page 321)
PS3	The prompt issued by **select** (page 460)
PS4	The bash debugging symbol (page 443)
PWD	The pathname of the working directory
REPLY	Holds the line that read accepts (page 490); also used by **select** (page 460)

Special Characters

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

Table 8-6 Shell special characters

Character	Use
NEWLINE	A control operator that initiates execution of a command (page 300)
;	A control operator that separates commands (page 300)
( )	A control operator that groups commands (page 302) for execution by a subshell; these characters are also used to identify a function (page 356)
(( ))	Evaluates an arithmetic expression (page 505)
&	A control operator that executes a command in the background (pages 150 and 300)
|	A control operator that sends standard output of the preceding command to standard input of the following command (pipeline; page 300)
|&	A control operator that sends standard output and standard error of the preceding command to standard input of the following command (page 293)
>	Redirects standard output (page 140)
>>	Appends standard output (page 144)
<	Redirects standard input (page 142)
<<	Here document (page 462)
*	Matches any string of zero or more characters in an ambiguous file reference (page 154)

Table 8-6 Shell special characters (continued)

<table>
<tr><th>Character</th><th>Use</th></tr>
<tr><td>?</td><td>Matches any single character in an ambiguous file reference (page 153)</td></tr>
<tr><td>\</td><td>Quotes the following character (page 50)</td></tr>
<tr><td>'</td><td>Quotes a string, preventing all substitution (page 50)</td></tr>
<tr><td>"</td><td>Quotes a string, allowing only variable and command substitution (pages 50 and 312)</td></tr>
<tr><td>`...`</td><td>Performs command substitution [deprecated, see $()]</td></tr>
<tr><td>[ ]</td><td>Character class in an ambiguous file reference (page 155)</td></tr>
<tr><td>$(( ))</td><td>Evaluates an arithmetic expression (page 369)</td></tr>
<tr><td>$</td><td>References a variable (page 310)</td></tr>
<tr><td>. (dot builtin)</td><td>Executes a command in the current shell (page 290)</td></tr>
<tr><td>#</td><td>Begins a comment (page 298)</td></tr>
<tr><td>{ }</td><td>Surrounds the contents of a function (page 356)</td></tr>
<tr><td>: (null builtin)</td><td>Returns true (page 498)</td></tr>
<tr><td>&& (Boolean AND)</td><td>A control operator that executes the command on the right only if the command on the left succeeds (returns a zero exit status; page 302)</td></tr>
<tr><td>|| (Boolean OR)</td><td>A control operator that executes the command on the right only if the command on the left fails (returns a nonzero exit status; page 302)</td></tr>
<tr><td>! (Boolean NOT)</td><td>Reverses the exit status of a command</td></tr>
<tr><td>$() (not in tcsh)</td><td>Performs command substitution (preferred form; page 371)</td></tr>
</table>

Locale

In conversational English, a *locale* is a place or location. When working with Linux, a locale specifies the way locale-aware programs display certain kinds of data such as times and dates, money and other numeric values, telephone numbers, and measurements. It can also specify collating sequence and printer paper size.

Localization and internationalization

Localization and internationalization go hand in hand: Internationalization is the process of making software portable to multiple locales while localization is the process of adapting software so that it meets the language, cultural, and other requirements of a specific locale. Linux is well internationalized so you can easily specify a locale for a given system or user. Linux uses variables to specify a locale.

i18n The term i18n is an abbreviation of the word *internationalization:* the letter *i* followed by 18 letters (*nternationalizatio*) followed by the letter *n*.

l10n The term l10n is an abbreviation of the word *localization:* the letter *l* followed by 10 letters (*ocalizatio*) followed by the letter *n*.

LC_: Locale Variables

The bash man page lists the following locale variables; other programs use additional locale variables. See the locale man pages (sections 1, 5, and 7) or use locale **––help** for more information.

- **LANG**—Specifies the locale category for categories not specified by an **LC_** variable (except see **LC_ALL**). Many setups use only this locale variable and do not specify any of the **LC_** variables.
- **LC_ALL**—Overrides the value of **LANG** and all other **LC_** variables.
- **LC_COLLATE**—Specifies the collating sequence for the sort utility (page 969) and for sorting the results of pathname expansion (page 313).
- **LC_CTYPE**—Specifies how characters are interpreted and how character classes within pathname expansion and pattern matching behave. Also affects the sort utility (page 969) when you specify the **–d** (**––dictionary-order**) or the **–i** (**––ignore-nonprinting**) options.
- **LC_MESSAGES**—Specifies how affirmative and negative answers appear and the language messages are displayed in.
- **LC_NUMERIC**—Specifies how numbers are formatted (e.g., are thousands separated by a comma or a period?).

Internationalized C programs call setlocale()

tip Internationalized C programs call **setlocale**(). Other languages have analogous facilities. Shell scripts are typically internationalized to the degree that the routines they call are. Without a call to **setlocale**(), the **hello, world** program will always display **hello, world**, regardless of how you set **LANG**.

You can set one or more of the **LC_** variables to a value using the syntax

xx_YY.CHARSET

where ***xx*** is the ISO-639 language code (e.g., **en** = English, **fr** = French, **zu** = Zulu), ***YY*** is the ISO-3166 country code (e.g., **FR** = France, **GF** = French Guiana, **PF** = French Polynesia), and ***CHARSET*** is the name of the character set (e.g., **UTF-8** [page 1131], **ASCII** [page 1083], **ISO-8859-1** [Western Europe], also called the *character map* or *charmap*). On some systems you can specify ***CHARSET*** using lowercase letters. For example, **en_GB.UTF-8** can specify English as written in Great Britain, **en_US.UTF-8** can specify English as written in the United States, and **fr_FR.UTF-8** can specify French as written in France.

The C locale

tip Setting the locale to C forces a program to process and display strings as the program was written (i.e., without translating input or output), which frequently means the program works in English. Many system scripts set **LANG** to **C** so they run in a known environment. Some text processing utilities run slightly faster when you set **LANG** to **C**. Setting **LANG** to **C** before you run sort can help ensure you get the results you expect.

If you want to make sure your shell script will work properly, put the following line near the top of the file:

```
export LANG=C
```

Following is an example of a difference that setting **LANG** can cause. It shows that having **LANG** set to different values can cause commands to behave differently, especially with regard to sorting.

```
$ echo $LANG
en_US.UTF-8
$ ls
m666  Makefile  merry
$ ls [l-n]*
m666  Makefile  merry

$ export LANG=C
$ ls
Makefile  m666  merry
$ ls [l-n]*
m666  merry
```

locale: Displays Locale Information

The locale utility displays information about the current and available locales. Without options, locale displays the value of the locale variables. In the following example, only the **LANG** variable is set, although you cannot determine this fact from the output. Unless explicitly set, each of the **LC_** variables derives its value from **LANG**.

```
$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

Typically, you will want all locale variables to have the same value. However, in some cases you might want to change the value of one or more locale variables. For example, if you are using paper size A4 but working in English, you could change the value of **LC_PAPER** to **nl_NL.utf8**.

The **–a** (all) option causes locale to display the names of available locales; **–v** (verbose; not in macOS) displays more complete information.

```
$ locale -av
locale: aa_DJ           archive: /usr/lib/locale/locale-archive
--------------------------------------------------------------------
    title | Afar language locale for Djibouti (CaduLaaqo Dialects).
   source | Ge'ez Frontier Foundation
  address | 7802 Solomon Seal Dr., Springfield, VA 22152, USA
    email | locales@geez.org
 language | aa
territory | DJ
 revision | 0.20
     date | 2003-07-05
  codeset | ISO-8859-1
...
```

The **–m** (maps) option causes locale to display the names of available character maps. On Linux systems, locale definition files are kept in the **/usr/share/i18n/locales** directory; on macOS systems, they are kept in **/usr/share/locale**.

Following are some examples of how some **LC_** variables change displayed values. Each of these command lines sets an **LC_** variable and places it in the environment of the utility it calls. The **+%x** format causes date to display the locale's date representation. The last example does not work under macOS.

```
$ LC_TIME=en_GB.UTF-8 date +%x
24/01/18
$ LC_TIME=en_US.UTF-8 date +%x
01/24/2018

$ ls xx
ls: impossible d'accéder à xx: Aucun fichier ou dossier de ce type
$ LC_MESSAGES=en_US.UTF-8 ls xx
ls: cannot access xx: No such file or directory
```

SETTING THE LOCALE

You might have to install a language package for a locale before you can specify a locale. If you are working in a GUI, it is usually easiest to change the locale using the GUI.

For all Linux distributions and macOS, put locale variable assignments in **~/.profile** or **~/.bash_profile** to affect both GUI and bash command-line logins for a single user. Remember to export the variables. The following line in one of these files will set all **LC_** variables for the given user to French as spoken in France:

```
export LANG=fr_FR.UTF-8
```

Under tcsh, put the following line in **~/.tcshrc** or **~/.cshrc** to have the same effect:

```
setenv LANG fr_FR.UTF-8
```

The following paragraphs explain how to use the command-line interface to change the locale for all users; the technique varies by distribution.

Fedora/RHEL Put locale variable assignments (previous page) in **/etc/profile.d/zlang.sh** (you will need to create this file; the filename was chosen to be executed after **lang.sh**) to affect both GUI and command-line logins for all users. Under tcsh, put the variable assignment in **/etc/profile.d/zlang.csh**.

Debian/Ubuntu/Mint Put locale variable assignments (previous page) in **/etc/default/locale** to affect both GUI and command-line logins for all users.

openSUSE Put locale variable assignments (previous page) in **/etc/profile.local** (you might need to create this file) to affect both GUI and command-line logins for all users. The **/etc/sysconfig/language** file controls the locale of GUI logins; see the file for instructions.

macOS Put locale variable assignments (previous page) in **/etc/profile** to affect both GUI and command-line logins for all users.

Time

UTC On networks with systems in different time zones it can be helpful to set all systems to the *UTC* (page 1131) time zone. Among other benefits, doing so can make it easier for an administrator to compare logged events on different systems over time. Each user account can be set to the local time for that user.

Time zone The time zone for a user is specified by an environment variable or, if one is not set, by the time zone for the system.

TZ The **TZ** variable gives a program access to information about the local time zone. This variable is typically set in a startup file (pages 288 and 382) and placed in the environment (page 480) so called programs have access to it. It has two syntaxes.

The first syntax of the **TZ** variable is

nam±val[nam2]

where ***nam*** is a string comprising three or more letters that typically name the time zone (e.g., PST; its value is not significant) and ***±val*** is the offset of the time zone from UTC, with positive values indicating the local time zone is west of the prime meridian and negative values indicating the local time zone is east of the prime meridian. If the ***nam2*** is present, it indicates the time zone observes daylight savings time; it is the name of the daylight savings time zone (e.g., PDT).

In the following example, date is called twice, once without setting the **TZ** variable and then with the **TZ** variable set in the environment in which date is called:

```
$ date
Wed May  3 10:08:06 PDT 2017

$ TZ=EST+5EDT date
Wed May  3 13:08:08 EDT 2017
```

The second syntax of the **TZ** variable is *continent/country*

where ***continent*** is the name of the continent or ocean and ***country*** is the name of the country that includes the desired time zone. This syntax points to a file in the **/usr/share/zoneinfo** hierarchy (next page). See tzselect (below) if you need help determining these values.

In the next example, date is called twice, once without setting the **TZ** variable and then with the **TZ** variable set in the environment in which date is called:

```
$ date
Wed May  3 10:09:27 PDT 2017

$ TZ=America/New_York date
Wed May  3 13:09:28 EDT 2017
```

See www.gnu.org/software/libc/manual/html_node/TZ-Variable.html for extensive documentation on the **TZ** variable.

tzconfig The tzconfig utility was available under Debian/Ubuntu and is now deprecated; use **dpkg-reconfigure tzdata** in its place.

tzselect The tzselect utility can help you determine the name of a time zone by asking you first to name the continent or ocean and then the country the time zone is in. If necessary, it asks for a time zone region (e.g., Pacific Time). This utility does not change system settings but rather displays a line telling you the name of the time zone. In the following example, the time zone is named **Europe/Paris**. Newer releases keep time zone information in **/usr/share/zoneinfo** (next page). Specifications such as **Europe/Paris** refer to the file in that directory (**/usr/share/zoneinfo/Europe/Paris**).

```
$ tzselect
Please identify a location so that time zone rules can be set correctly.
Please select a continent or ocean.
 1) Africa
...
 8) Europe
 9) Indian Ocean
10) Pacific Ocean
11) none - I want to specify the time zone using the Posix TZ format.
#? 8
Please select a country.
 1) Aaland Islands       18) Greece                35) Norway
...
15) France               32) Monaco                49) Vatican City
16) Germany              33) Montenegro
17) Gibraltar            34) Netherlands
#? 15
...
Here is that TZ value again, this time on standard output so that you
can use the /usr/bin/tzselect command in shell scripts:
Europe/Paris
```

/etc/timezone Under some distributions, including Debian/Ubuntu/Mint, the **/etc/timezone** file holds the name of the local time zone.

```
$ cat /etc/timezone
America/Los_Angeles
```

/usr/share/zoneinfo The **/usr/share/zoneinfo** directory hierarchy holds time zone data files. Some time zones are held in regular files in the **zoneinfo** directory (e.g., Japan and GB) while others are held in subdirectories (e.g., Azores and Pacific). The following example shows a small part of the **/usr/share/zoneinfo** directory hierarchy and illustrates how file (page 820) reports on a time zone file.

```
$ find /usr/share/zoneinfo
/usr/share/zoneinfo
/usr/share/zoneinfo/Atlantic
/usr/share/zoneinfo/Atlantic/Azores
/usr/share/zoneinfo/Atlantic/Madeira
/usr/share/zoneinfo/Atlantic/Jan_Mayen
...
/usr/share/zoneinfo/Japan
/usr/share/zoneinfo/GB
/usr/share/zoneinfo/US
/usr/share/zoneinfo/US/Pacific
/usr/share/zoneinfo/US/Arizona
/usr/share/zoneinfo/US/Michigan
...

$ file /usr/share/zoneinfo/Atlantic/Azores
/usr/share/zoneinfo/Atlantic/Azores: timezone data, version 2, 12 gmt
time flags, 12 std time flags, no leap seconds, 220 transition times, 12
abbreviation chars
```

/etc/localtime Some Linux distributions use a link at **/etc/localtime** to a file in **/usr/share/zoneinfo** to specify the local time zone. Others copy the file from the **zoneinfo** directory to **localtime**. Following is an example of setting up this link; to create this link you must work with **root** privileges.

```
# date
Wed Tue Jan 24 13:55:00 PST 2018
# cd /etc
# ln -sf /usr/share/zoneinfo/Europe/Paris localtime
# date
Wed Jan 24 22:55:38 CET 2018
```

On some of these systems, the **/etc/systemconfig/clock** file sets the **ZONE** variable to the name of the time zone:

```
$ cat /etc/sysconfig/clock
# The time zone of the system is defined by the contents of /etc/localtime.
# This file is only for evaluation by system-config-date, do not rely on its
# contents elsewhere.
ZONE="Europe/Paris"
```

macOS On macOS, you can use systemsetup to work with the time zone.

```
$ systemsetup -gettimezone
Time Zone: America/Los_Angeles

$ systemsetup -listtimezones
Time Zones:
 Africa/Abidjan
 Africa/Accra
 Africa/Addis_Ababa
...

$ systemsetup -settimezone America/Los_Angeles
Set TimeZone: America/Los_Angeles
```

Processes

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

Process Structure

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

init daemon A Linux system begins execution by starting the **init** daemon, a single process called a *spontaneous process,* with PID number 1. This process holds the same position in the process structure as the root directory does in the file structure: It is the ancestor of all processes the system and users work with. When a command-line system is in multiuser mode, init runs getty or mingetty processes, which display **login:** prompts on terminals and virtual consoles. When a user responds to the prompt and presses RETURN, getty or mingetty passes control to a utility named login, which checks the username and password combination. After the user logs in, the login process becomes the user's shell process.

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

Process Identification

PID numbers Linux assigns a unique PID (process identification) number at the inception of each process. As long as a process exists, it keeps the same PID number. During one session the same process is always executing the login shell (page 288). When you fork a new process—for example, when you use an editor—the PID number of the new (child) process is different from that of its parent process. When you return to the login shell, it is still being executed by the same process and has the same PID number as when you logged in.

The following example shows that the process running the shell forked (is the parent of) the process running ps. When you call it with the **–f** option, ps displays a full listing of information about each process. The line of the ps display with **bash** in the **CMD** column refers to the process running the shell. The column headed by **PID** identifies the PID number. The column headed by **PPID** identifies the PID number of the *parent* of the process. From the PID and PPID columns you can see that the process running the shell (PID 21341) is the parent of the processes running sleep (PID 22789) and ps (PID 22790).

```
$ sleep 10 &
[1] 22789
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
max      21341 21340  0 10:42 pts/16   00:00:00 bash
max      22789 21341  0 17:30 pts/16   00:00:00 sleep 10
max      22790 21341  0 17:30 pts/16   00:00:00 ps -f
```

Refer to page 946 for more information on ps and the columns it displays when you specify the **–f** option. A second pair of **sleep** and **ps –f** commands shows that the shell is still being run by the same process but that it forked another process to run sleep:

```
$ sleep 10 &
[1] 22791
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
max      21341 21340  0 10:42 pts/16   00:00:00 bash
max      22791 21341  0 17:31 pts/16   00:00:00 sleep 10
max      22792 21341  0 17:31 pts/16   00:00:00 ps -f
```

You can also use pstree (or **ps ––forest**, with or without the **–e** option) to see the parent–child relationship of processes. The next example shows the **–p** option to pstree, which causes it to display PID numbers:

```
$ pstree -p
systemd(1)-+-NetworkManager(655)---{NetworkManager}(702)
           |-abrtd(657)---abrt-dump-oops(696)
```

```
              |-accounts-daemon(1204)---{accounts-daemo}(1206)
              |-agetty(979)
...
              |-login(984)---bash(2071)-+-pstree(2095)
              |                         `-sleep(2094)
...
```

The preceding output is abbreviated. The first line shows the PID 1 (**systemd init**) and a few of the processes it is running. The line that starts with **–login** shows a textual user running sleep in the background and pstree in the foreground. The tree for a user running a GUI is much more complex. Refer to "$$: PID Number" on page 476 for a description of how to instruct the shell to report on PID numbers.

EXECUTING A COMMAND

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

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

Builtins Although the shell forks a process to run most commands, some commands are built into the shell (e.g., cd, alias, jobs, pwd). The shell does not fork a process to run builtins. For more information refer to "Builtins" on page 157.

Variables Within a given process, such as a login shell or subshell, you can declare, initialize, read, and change variables. Some variables, called shell variables, are local to a process. Other variables, called environment variables, are available to child processes. For more information refer to "Variables" on page 479.

Hash table The first time you specify a command as a simple filename (and not a relative or absolute pathname), the shell looks in the directories specified by the **PATH** (bash; page 318) or **path** (tcsh; page 403) variable to find that file. When it finds the file, the shell records the absolute pathname of the file in its hash table. When you give the command again, the shell finds it in its hash table, saving the time needed to search through the directories in **PATH**. The shell deletes the hash table when you log out and starts a new hash table when you start a session. This section shows some of the ways you can use the bash hash builtin; tcsh uses different commands for working with its hash table.

When you call the hash builtin without any arguments, it displays the hash table. When you first log in, the hash table is empty:

```
$ hash
hash: hash table empty
$ who am i
sam      pts/2        2017-03-09 14:24 (plum)
$ hash
hits    command
   1    /usr/bin/who
```

The hash –r option causes bash to empty the hash table, as though you had just logged in; tcsh uses rehash for a similar purpose.

```
$ hash -r
$ hash
hash: hash table empty
```

Having bash empty its hash table is useful when you move a program to a different directory in **PATH** and bash cannot find the program in its new location, or when you have two programs with the same name and bash is calling the wrong one. Refer to the bash info page for more information on the hash builtin.

History

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

history **can help track down mistakes**

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

The history builtin displays the history list. If it does not, read the next section, which describes the variables you might need to set.

Variables That Control History

The TC Shell's history mechanism is similar to bash's but uses different variables and has some other differences. See page 384 for more information.

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

When you exit from the shell, the most recently executed commands are saved in the file whose name is stored in the **HISTFILE** variable (default is **~/.bash_history**). The next time you start the shell, this file initializes the history list. The value of the **HISTFILESIZE** variable determines the number of lines of history saved in **HISTFILE** (see Table 8-7).

Table 8-7 History variables

Variable	Default	Function
HISTSIZE	1,000 events	Maximum number of events saved during a session
HISTFILE	**~/.bash_history**	Location of the history file
HISTFILESIZE	1,000–2,000 events	Maximum number of events saved between sessions

Event number The Bourne Again Shell assigns a sequential *event number* to each command line. You can display this event number as part of the bash prompt by including **\!** in **PS1** (page 319). Examples in this section show numbered prompts when they help to illustrate the behavior of a command.

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

```
$ HISTSIZE=100
```

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

```
$ HISTFILESIZE=100
```

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

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

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

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

Handy history aliases

tip Creating the following aliases makes working with history easier. The first allows you to give the command **h** to display the ten most recent events. The second alias causes the command **hg *string*** to display all events in the history list that contain ***string***. Put these aliases in your **~/.bashrc** file to make them available each time you log in. See page 352 for more information on aliases.

```
$ alias 'h=history | tail'
$ alias 'hg=history | grep'
```

Reexecuting and Editing Commands

You can reexecute any event in the history list. Not having to reenter long command lines allows you to reexecute events more easily, quickly, and accurately than you could if you had to retype the command line in its entirety. You can recall, modify, and reexecute previously executed events in three ways: You can use the fc builtin (next), the exclamation point commands (page 341), or the Readline Library, which uses a one-line vi- or emacs-like editor to edit and execute events (page 345).

Which method to use?

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

fc: Displays, Edits, and Reexecutes Commands

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

Viewing the History List

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

```
$ fc -l
1024     cd
1025     view calendar
1026     vim letter.adams01
1027     aspell -c letter.adams01
1028     vim letter.adams01
1029     lpr letter.adams01
1030     cd ../memos
1031     ls
```

```
1032     rm *0405
1033     fc -l
1034     cd
1035     whereis aspell
1036     man aspell
1037     cd /usr/share/doc/*aspell*
1038     pwd
1039     ls
1040     ls man-html
```

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

*fc –l [**first** [**last**]]*

The fc builtin lists commands beginning with the most recent event that matches ***first***. The argument can be an event number, the first few characters of the command line, or a negative number, which specifies the ***n***th previous command. Without ***last***, fc displays events through the most recent. If you include ***last***, fc displays commands from the most recent event that matches ***first*** through the most recent event that matches ***last***.

The next command displays the history list from event 1030 through event 1035:

```
$ fc -l 1030 1035
1030     cd ../memos
1031     ls
1032     rm *0405
1033     fc -l
1034     cd
1035     whereis aspell
```

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

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

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

```
$ fc -l 1027 1027
1027     aspell -c letter.adams01
```

Editing and Reexecuting Previous Commands

You can use fc to edit and reexecute previous commands.

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

When you call fc with the **–e** option followed by the name of an editor, fc calls the editor with event(s) in the Work buffer. By default, fc invokes the vi(m) or nano editor. Without ***first*** and ***last***, it defaults to the most recent command. The next example invokes the vim editor (Chapter 6) to edit the most recent command. When you exit from the editor, the shell executes the command.

```
$ fc -e vi
```

The fc builtin uses the stand-alone vim editor. If you set the **EDITOR** variable, you do not need to use the **–e** option to specify an editor on the command line. Because the value of **EDITOR** has been changed to **/usr/bin/emacs** and fc has no arguments, the following command edits the most recent command using the emacs editor (Chapter 7):

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

If you call it with a single argument, fc invokes the editor on the specified command. The following example starts the editor with event 1029 in the Work buffer:

```
$ fc 1029
```

As described earlier, you can identify commands either by using numbers or by specifying the first few characters of the command name. The following example calls the editor to work on events from the most recent event that begins with the letters **vim** through event 1030:

```
$ fc vim 1030
```

Clean up the fc buffer

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

Reexecuting Commands Without Calling the Editor

You can also reexecute previous commands without using an editor. If you call fc with the **–s** option, it skips the editing phase and reexecutes the command. The following example reexecutes event 1029:

```
$ fc -s 1029
lpr letter.adams01
```

The next example reexecutes the previous command:

```
$ fc -s
```

When you reexecute a command, you can tell fc to substitute one string for another. The next example substitutes the string **john** for the string **adams** in event 1029 and executes the modified event:

```
$ fc -s adams=john 1029
lpr letter.john01
```

USING AN EXCLAMATION POINT (!) TO REFERENCE EVENTS

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

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

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

EVENT DESIGNATORS

An event designator specifies a command in the history list. Table 8-8 lists event designators.

Table 8-8 Event designators

Designator	Meaning
!	Starts a history event unless followed immediately by SPACE, NEWLINE, **=**, or **(**.
!!	The previous command.
!***n***	Command number ***n*** in the history list.
!–***n***	The ***n***th preceding command.
!***string***	The most recent command line that started with ***string***.
!?***string***[?]	The most recent command that contained ***string***. The last **?** is optional.
!#	The current command (as you have it typed so far).

!! reexecutes the previous event

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

```
44 $ ls -l text
-rw-rw-r--. 1 max pubs 45 04-30 14:53 text
45 $ !!
ls -l text
-rw-rw-r--. 1 max pubs 45 04-30 14:53 text
```

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

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

```
51 $ !44
ls -l text
-rw-rw-r--. 1 max pubs 45 04-30 14:53 text
52 $ !-8
ls -l text
-rw-rw-r--. 1 max pubs 45 04-30 14:53 text
```

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

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

optional

WORD DESIGNATORS

A *word designator* specifies a word (token) or series of words from an event (a command line). Table 8-9 on page 344 lists word designators. The words on a command line are numbered starting with 0 (the first word, usually the command), continuing with 1 (the first word following the command), and ending with ***n*** (the last word on the command line).

To specify a particular word from a previous event, follow the event designator (such as **!14**) with a colon and the number of the word in the previous event. For example,

!14:3 specifies the third word following the command from event 14. You can specify the first word following the command (word number 1) using a caret (^) and the last word using a dollar sign ($). You can specify a range of words by separating two word designators with a hyphen.

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

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

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

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

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

Table 8-9 Word designators

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

Modifiers

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

Substitute modifier The following example shows the *substitute modifier* correcting a typo in the previous event:

```
$ car /home/zach/memo.0507 /home/max/letter.0507
bash: car: command not found
$ !!:s/car/cat
cat /home/zach/memo.0507 /home/max/letter.0507
...
```

The substitute modifier has the syntax

[g]s/***old***/***new***/

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

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

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

produces the same results as

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

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

```
$ ^car^cat
cat /home/zach/memo.0507 /home/max/letter.0507
...
```

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

Other modifiers

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

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

```
$ ls /etc/ssh/ssh_config
/etc/ssh/ssh_config
$ !!:p
ls /etc/ssh/ssh_config
$ !!:h:p
ls /etc/ssh
```

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

Table 8-10 Event modifiers

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

The Readline Library

Command-line editing under the Bourne Again Shell is implemented through the *Readline Library*, which is available to any application written in C. Any application that uses the Readline Library supports line editing that is consistent with that provided by bash. Programs that use the Readline Library, including bash, read **~/.inputrc** (page 349) for key binding information and configuration settings. The **––noediting** command-line option turns off command-line editing in bash.

vi mode You can choose one of two editing modes when using the Readline Library in bash: emacs or vi(m). Both modes provide many of the commands available in the stand-alone versions of the emacs and vim editors. You can also use the ARROW keys to move around. Up and down movements move you backward and forward through the history list. In addition, Readline provides several types of interactive word completion (page 348). The default mode is emacs; you can switch to vi mode using the following command:

```
$ set -o vi
```

emacs mode The next command switches back to emacs mode:

```
$ set -o emacs
```

vi Editing Mode

Before you start, make sure the shell is in vi mode.

When you enter bash commands while in vi editing mode, you are in Input mode (page 169). As you enter a command, if you discover an error before you press RETURN, you can press ESCAPE to switch to vim Command mode. This setup is different from the stand-alone vim editor's initial mode. While in Command mode you can use many vim commands to edit the command line. It is as though you were using vim to edit a copy of the history file with a screen that has room for only one command. When you use the **k** command or the UP ARROW to move up a line, you access the previous command. If you then use the **j** command or the DOWN ARROW to move down a line, you return to the original command. To use the **k** and **j** keys to move between commands, you must be in Command mode; you can use the ARROW keys in both Command and Input modes.

The command-line vim editor starts in Input mode

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

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

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

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

When the command you want to edit is displayed, you can modify the command line using vim Command mode editing commands such as **x** (delete character), **r** (replace character), **~** (change case), and **.** (repeat last change). To switch to Input mode, use an Insert (**i**, **I**), Append (**a**, **A**), Replace (**R**), or Change (**c**, **C**) command. You do not have to return to Command mode to execute a command; simply press RETURN, even if the cursor is in the middle of the command line. For more information refer to the vim tutorial on page 167. Refer to page 213 for a summary of vim commands.

emacs EDITING MODE

Unlike the vim editor, emacs is modeless. You need not switch between Command mode and Input mode because most emacs commands are control characters (page 231), allowing emacs to distinguish between input and commands. Like vim, the emacs command-line editor provides commands for moving the cursor on the command line and through the command history list and for modifying part or all of a command. However, in a few cases, the emacs command-line editor commands differ from those used in the stand-alone emacs editor.

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

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

You can add text to the command line by moving the cursor to the position you want to enter text and typing. To delete text, move the cursor just to the right of the characters you want to delete and press the erase key (page 29) once for each character you want to delete.

CONTROL-D can terminate your screen session

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

If you want to delete the entire command line, press the line kill key (page 30). You can press this key while the cursor is anywhere in the command line. Use CONTROL-K to delete from the cursor to the end of the line. Refer to page 270 for a summary of emacs commands.

Readline Completion Commands

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

Command Completion

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

In the following example, the user types **bz** and presses TAB. The shell beeps (the user is in emacs mode) to indicate that several commands start with the letters **bz**. The user enters another TAB to cause the shell to display a list of commands that start with **bz** followed by the command line as the user has entered it so far:

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

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

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

Pathname Completion

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

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

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

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

```
$ cat films/dark_ ⇨ TAB (beep) ⇨ TAB
dark_passage  dark_victory
$ cat films/dark_█
```

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

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

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

Variable Completion

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

```
$ echo $HO ⇨ TAB (beep) ⇨ TAB
$HOME      $HOSTNAME  $HOSTTYPE
$ echo $HOM ⇨ TAB ⇨ E
```

Pressing RETURN executes the command

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

.inputrc: Configuring the Readline Library

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

Variables

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

*set **variable value***

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

Table 8-11 Readline variables

Variable	Effect
editing-mode	Set to **vi** to start Readline in vi mode. Set to **emacs** to start Readline in emacs mode (the default). Similar to the **set –o vi** and **set –o emacs** shell commands (page 346).
horizontal-scroll-mode	Set to **on** to cause long lines to extend off the right edge of the display area. Moving the cursor to the right when it is at the right edge of the display area shifts the line to the left so you can see more of the line. Shift the line back by moving the cursor back past the left edge. The default value is **off**, which causes long lines to wrap onto multiple lines of the display.
mark-directories	Set to **off** to cause Readline not to place a slash (/) at the end of directory names it completes. The default value is **on**.
mark-modified-lines	Set to **on** to cause Readline to precede modified history lines with an asterisk. The default value is **off**.

Key Bindings

You can map keystroke sequences to Readline commands, changing or extending the default bindings. Like the emacs editor, the Readline Library includes many commands that are not bound to a keystroke sequence. To use an unbound command, you must map it using one of the following forms:

> ***keyname*****:** ***command_name***
> **"** ***keystroke_sequence*** **" :** ***command_name***

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

In the second form, you specify a string that describes a sequence of keys that will be bound to the command. You can use the emacs-style backslash escape sequences (page 231) to represent the special keys CONTROL (**\C**), META (**\M**), and ESCAPE (**\e**). Specify a backslash by escaping it with another backslash: ****. Similarly, a double or single quotation mark can be escaped with a backslash: **\"** or **\'**.

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

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

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

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

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

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

```
"QQ": "The Linux Operating System"
```

This command causes bash to insert the string **The Linux Operating System** when you type **QQ** on the command line.

Conditional Constructs

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

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

where ***test*** is **mode**, **term**, or a program name such as **bash**. If ***test*** equals ***value*** (or if ***test*** is *true* when ***value*** is not specified), this structure executes the first set of ***commands***. If ***test*** does not equal ***value*** (or if ***test*** is *false* when ***value*** is not specified), it executes the second set of ***commands*** if they are present or exits from the structure if they are not present.

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

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

   ```
   $if mode=vi
   ```

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

2. You can test the type of terminal.

   ```
   $if term=xterm
   ```

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

3. You can test the application name.

   ```
   $if bash
   ```

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

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

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

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

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

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

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

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

ALIASES

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

Under bash the syntax of the alias builtin is

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

Under tcsh the syntax is

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

In the bash syntax no SPACEs are permitted around the equal sign. If ***value*** contains SPACEs or TABs, you must enclose ***value*** within quotation marks. Unlike aliases under tcsh, a bash alias does not accept an argument from the command line in ***value***. Use a bash function (page 356) when you need to use an argument.

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

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

You can nest aliases. Aliases are disabled for noninteractive shells (that is, shell scripts). Use the unalias builtin to remove an alias. When you give an alias builtin command without any arguments, the shell displays a list of all defined aliases:

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

To view the alias for a particular name, enter the command **alias** followed by the name of the alias. Most Linux distributions define at least some aliases. Enter an **alias** command to see which aliases are in effect. You can delete the aliases you do not want from the appropriate startup file.

Single Versus Double Quotation Marks in Aliases

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

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

```
$ echo $PWD
/home/max
$ alias dirA="echo Working directory is $PWD"
$ alias dirA
alias dirA='echo Working directory is /home/max'
```

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

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

After creating the **dirA** and **dirB** aliases, Max uses cd to make **cars** his working directory and gives each of the aliases as a command. The alias he created using double quotation marks displays the name of the directory he created the alias in as the working directory (which is wrong). In contrast, the **dirB** alias displays the proper name of the working directory:

```
$ cd cars
$ dirA
Working directory is /home/max
$ dirB
Working directory is /home/max/cars
```

How to prevent the shell from invoking an alias

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

Examples of Aliases

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

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

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

```
$ alias l='ls -ltr'
$ l
-rw-r-----. 1 max pubs  3089 02-11 16:24 XTerm.ad
-rw-r--r--. 1 max pubs 30015 03-01 14:24 flute.ps
-rw-r--r--. 1 max pubs   641 04-01 08:12 fixtax.icn
-rw-r--r--. 1 max pubs   484 04-09 08:14 maptax.icn
drwxrwxr-x. 2 max pubs  1024 08-09 17:41 Tiger
drwxrwxr-x. 2 max pubs  1024 09-10 11:32 testdir
-rwxr-xr-x. 1 max pubs   485 09-21 08:03 floor
drwxrwxr-x. 2 max pubs  1024 09-27 20:19 Test_Emacs
```

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

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

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

The aliases in the next example cause the shell to substitute **ls –l** each time you give an **ll** command and **ls –F** each time you use **ls**. The **–F** option causes ls to print a slash (**/**) at the end of directory names and an asterisk (*) at the end of the names of executable files.

```
$ alias ls='ls -F'
$ alias ll='ls -l'
$ ll
drwxrwxr-x. 2 max pubs  1024 09-27 20:19 Test_Emacs/
drwxrwxr-x. 2 max pubs  1024 08-09 17:41 Tiger/
-rw-r-----. 1 max pubs  3089 02-11 16:24 XTerm.ad
-rw-r--r--. 1 max pubs   641 04-01 08:12 fixtax.icn
-rw-r--r--. 1 max pubs 30015 03-01 14:24 flute.ps
-rwxr-xr-x. 1 max pubs   485 09-21 08:03 floor*
-rw-r--r--. 1 max pubs   484 04-09 08:14 maptax.icn
drwxrwxr-x. 2 max pubs  1024 09-10 11:32 testdir/
```

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

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

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

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

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

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

```
$ ll f*
-rw-r--r--. 1 max pubs   641 04-01 08:12 fixtax.icn
-rw-r--r--. 1 max pubs 30015 03-01 14:24 flute.ps
-rwxr-xr-x. 1 max pubs   485 09-21 08:03 floor*
```

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

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

Functions

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

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

Removing variables and functions that have the same name

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

The syntax that declares a shell function is

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

where the word *function* is optional (and is frequently omitted; it is not portable), ***function-name*** is the name you use to call the function, and ***commands*** comprise the list of commands the function executes when you call it. The ***commands*** can be anything you would include in a shell script, including calls to other functions.

The opening brace ({) can appear on the line following the function name. Aliases and variables are expanded when a function is read, not when it is executed. You can use the **break** statement (page 453) within a function to terminate its execution.

You can declare a function on a single line. Because the closing brace must appear as a separate command, you must place a semicolon before the closing brace when you use this syntax:

```
$ say_hi() { echo "hi" ; }
$ say_hi
hi
```

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

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

The next example creates a simple function that displays the date, a header, and a list of the people who are logged in on the system. This function runs the same commands as the **whoson** script described on page 295. In this example the function is being entered from the keyboard. The greater than (**>**) signs are secondary shell prompts (**PS2**); do not enter them.

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

$ whoson
Thurs Aug 9 15:44:58 PDT 2018
Users Currently Logged On
hls      console      2018-08-08 08:59  (:0)
max      pts/4        2018-08-08 09:33  (0.0)
zach     pts/7        2018-08-08 09:23  (guava)
```

Function local variables

You can use the local builtin only within a function. This builtin causes its arguments to be local to the function it is called from and its children. Without local, variables declared in a function are available to the shell that called the function (functions are run in the shell they are called from). The following function demonstrates the use of local:

```
$ demo () {
> x=4
> local y=8
> echo "demo: $x $y"
> }
$ demo
demo: 4 8
$ echo $x
4
$ echo $y

$
```

The **demo** function, which is entered from the keyboard, declares two variables, **x** and **y**, and displays their values. The variable **x** is declared with a normal assignment statement while **y** is declared using local. After running the function, the shell that called the function has access to **x** but knows nothing of **y**. See page 488 for another example of function local variables.

Export a function

An **export –f** command places the named function in the environment so it is available to child processes.

Functions in startup files

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

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

$ . ~/.bash_profile
```

You can specify arguments when you call a function. Within the function these arguments are available as positional parameters (page 470). The following example shows the **arg1** function entered from the keyboard:

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

See the function **switch** () on page 290 for another example of a function.

optional

The following function allows you to place variables in the environment (export them) using tcsh syntax. The env utility lists all environment variables and their values and verifies that **setenv** worked correctly:

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

eval

The **$#** special parameter (page 475) takes on the value of the number of command-line arguments. This function uses the eval builtin to force bash to scan the command **$1=$2** *twice*. Because **$1=$2** begins with a dollar sign ($), the shell treats the entire string as a single token—a command. With variable substitution performed, the command name becomes **TCL_LIBRARY=/usr/local/lib/tcl**, which results in an error.

With eval, a second scanning splits the string into the three desired tokens, and the correct assignment occurs. See page 500 for more information on eval.

CONTROLLING bash: FEATURES AND OPTIONS

This section explains how to control bash features and options using command-line options and the set and shopt builtins. The shell sets flags to indicate which options are set (on) and expands $– to a list of flags that are set; see page 478 for more information.

bash COMMAND-LINE OPTIONS

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

Table 8-12 bash command-line options

Option	Explanation	Syntax
Help	Displays a usage message.	**––help**
No edit	Prevents users from using the Readline Library (page 345) to edit command lines in an interactive shell.	**––noediting**
No profile	Prevents reading these startup files (page 288): **/etc/profile**, **~/.bash_profile**, **~/.bash_login**, and **~/.profile**.	**––noprofile**
No rc	Prevents reading the **~/.bashrc** startup file (page 289). This option is on by default if the shell is called as **sh**.	**––norc**
POSIX	Runs bash in POSIX mode.	**––posix**
Version	Displays bash version information and exits.	**––version**
Login	Causes bash to run as though it were a login shell.	**–l** (lowercase "l")
shopt	Runs a shell with the ***opt*** shopt option (page 360). A **–O** (uppercase "O") sets the option; **+O** unsets it.	[±]**O** [***opt***]
End of options	On the command line, signals the end of options. Subsequent tokens are treated as arguments even if they begin with a hyphen (**–**).	**––**

Shell Features

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

Features, options, variables, attributes?

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

set ±o: Turns Shell Features On and Off

The set builtin, when used with the **–o** or **+o** option, enables, disables, and lists certain bash features (the set builtin in tcsh works differently). For example, the following command turns on the **noclobber** feature (page 143):

```
$ set -o noclobber
```

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

```
$ set +o noclobber
```

The command **set –o** without an option lists each of the features controlled by set, followed by its state (on or off). The command **set +o** without an option lists the same features in a form you can use as input to the shell. Table 8-13 lists bash features. This table does not list the **–i** option because you cannot set it. The shell sets this option when it is invoked as an interactive shell. See page 472 for a discussion of other uses of set.

shopt: Turns Shell Features On and Off

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

```
$ shopt -s dotglob
```

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

```
$ shopt -u dotglob
```

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

```
$ shopt dotglob
dotglob         off
```

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

Setting set ±o features using shopt

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

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

Table 8-13 bash features

Feature	Description	Syntax	Alternative syntax
allexport	Automatically places in the environment (exports) all variables and functions you create or modify after giving this command (default is off).	**set –o allexport**	**set –a**
braceexpand	Causes bash to perform brace expansion (default is on; page 366).	**set –o braceexpand**	**set –B**
cdspell	Corrects minor spelling errors in directory names used as arguments to cd (default is off).	**shopt –s cdspell**	
cmdhist	Saves all lines of a multiline command in the same history entry, adding semicolons as needed (default is on).	**shopt –s cmdhist**	
dotglob	Causes shell special characters (wildcards; page 152) in an ambiguous file reference to match a leading period in a filename. By default, special characters do not match a leading period: You must always specify the filenames **.** and **..** explicitly because no pattern ever matches them (default is off).	**shopt –s dotglob**	
emacs	Specifies emacs editing mode for command-line editing (default is on; page 347).	**set –o emacs**	
errexit	Causes bash to exit when a pipeline (page 145), which can be a simple command (page 133; not a control structure), fails (default is off).	**set –o errexit**	**set –e**

Table 8-13 bash features (continued)

Feature	Description	Syntax	Alternative syntax
execfail	Causes a shell script to continue running when it cannot find the file that is given as an argument to exec. By default, a script terminates when exec cannot find the file that is given as its argument (default is off).	**shopt –s execfail**	
expand_aliases	Causes aliases (page 352) to be expanded (default is on for interactive shells and off for noninteractive shells).	**shopt –s expand_aliases**	
hashall	Causes bash to remember where commands it has found using **PATH** (page 318) are located (default is on).	**set –o hashall**	**set –h**
histappend	Causes bash to append the history list to the file named by **HISTFILE** (page 336) when the shell exits (default is off [bash overwrites this file]).	**shopt –s histappend**	
histexpand	Turns on the history mechanism (which uses exclamation points by default; page 341). Turn this feature off to turn off history expansion (default is on).	**set –o histexpand**	**set –H**
history	Enables command history (default is on; page 336).	**set –o history**	
huponexit	Specifies that bash send a SIGHUP signal to all jobs when an interactive login shell exits (default is off).	**shopt –s huponexit**	
ignoreeof	Specifies that bash must receive ten EOF characters before it exits. Useful on noisy dial-up lines (default is off).	**set –o ignoreeof**	
monitor	Enables job control (default is on; page 304).	**set –o monitor**	**set –m**
nocaseglob	Causes ambiguous file references (page 152) to match filenames without regard to case (default is off).	**shopt –s nocaseglob**	
noclobber	Helps prevent overwriting files (default is off; page 143).	**set –o noclobber**	**set –C**
noglob	Disables pathname expansion (default is off; page 152).	**set –o noglob**	**set –f**

Table 8-13 bash features (continued)

Feature	Description	Syntax	Alternative syntax
notify	With job control (page 304) enabled, reports the termination status of background jobs immediately (default is off: bash displays the status just before the next prompt).	**set –o notify**	**set –b**
nounset	Displays an error when the shell tries to expand an unset variable; bash exits from a script but not from an interactive shell (default is off: bash substitutes a null value for an unset variable).	**set –o nounset**	**set –u**
nullglob	Causes bash to substitute a null string for ambiguous file references (page 152) that do not match a filename (default is off: bash passes these file references as is).	**shopt –s nullglob**	
pipefail	Sets the exit status of a pipeline to the exit status of the last (rightmost) simple command that failed (returned a nonzero exit status) in the pipeline; if no command failed, exit status is set to zero (default is off: bash sets the exit status of a pipeline to the exit status of the final command in the pipeline).	**set –o pipefail**	
posix	Runs bash in POSIX mode (default is off).	**set –o posix**	
verbose	Displays each command line after bash reads it but before bash expands it (default is off). See also **xtrace**.	**set –o verbose**	**set –v**
vi	Specifies vi editing mode for command-line editing (default is off; page 346).	**set –o vi**	
xpg_echo	Causes the echo builtin to expand backslash escape sequences without the need for the **–e** option (default is off; page 457).	**shopt –s xpg_echo**	
xtrace	Turns on shell debugging: Displays the value of **PS4** (page 321) followed by each input line after the shell reads and expands it (default is off; see page 442 for a discussion). See also **verbose**.	**set –o xtrace**	**set –x**

Processing the Command Line

Whether you are working interactively or running a shell script, bash needs to read a command line before it can start processing it—bash always reads at least one line before processing a command. Some bash builtins, such as **if** and **case**, as well as functions and quoted strings, span multiple lines. When bash recognizes a command that covers more than one line, it reads the entire command before processing it. In interactive sessions, bash prompts with the secondary prompt (**PS2**, **>** by default; page 321) as you type each line of a multiline command until it recognizes the end of the command:

```
$ ps -ef |
> grep emacs
zach     26880 24579  1 14:42 pts/10   00:00:00 emacs notes
zach     26890 24579  0 14:42 pts/10   00:00:00 grep emacs

$ function hello () {
> echo hello there
> }
$
```

For more information refer to "Implicit Command-Line Continuation" on page 512. After reading a command line, bash applies history expansion and alias substitution to the command line.

History Expansion

"Reexecuting and Editing Commands" on page 338 discusses the commands you can give to modify and reexecute command lines from the history list. History expansion is the process bash uses to turn a history command into an executable command line. For example, when you enter the command **!!**, history expansion changes that command line so it is the same as the previous one. History expansion is turned on by default for interactive shells; **set +o histexpand** turns it off. History expansion does not apply to noninteractive shells (shell scripts).

Alias Substitution

Aliases (page 352) substitute a string for the first word of a simple command. By default, alias substitution is turned on for interactive shells and off for noninteractive shells; **shopt –u expand_aliases** turns it off.

Parsing and Scanning the Command Line

After processing history commands and aliases, bash does not execute the command immediately. One of the first things the shell does is to *parse* (isolate strings of characters in) the command line into tokens (words). After separating tokens and before executing the command, the shell scans the tokens and performs *command-line expansion*.

Command-Line Expansion

Both interactive and noninteractive shells transform the command line using *command-line expansion* before passing the command line to the program being called. You can use a shell without knowing much about command-line expansion, but you can use what a shell has to offer to a better advantage with an understanding of this topic. This section covers Bourne Again Shell command-line expansion; TC Shell command-line expansion is covered starting on page 384.

The Bourne Again Shell scans each token for the various types of expansion and substitution in the following order. Most of these processes expand a word into a single word. Only brace expansion, word splitting, and pathname expansion can change the number of words in a command (except for the expansion of the variable **"$@"**—see page 474).

1. Brace expansion (next page)
2. Tilde expansion (page 368)
3. Parameter and variable expansion (page 368)
4. Arithmetic expansion (page 369)
5. Command substitution (page 371)
6. Word splitting (page 372)
7. Pathname expansion (page 372)
8. Process substitution (page 374)
9. Quote removal (page 374)

Order of Expansion

The order in which bash carries out these steps affects the interpretation of commands. For example, if you set a variable to a value that looks like the instruction for output redirection and then enter a command that uses the variable's value to perform redirection, you might expect bash to redirect the output.

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

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

Quotation marks can alter expansion

tip Double and single quotation marks cause the shell to behave differently when performing expansions. Double quotation marks permit parameter and variable expansion but suppress other types of expansion. Single quotation marks suppress all types of expansion.

Brace Expansion

Brace expansion, which originated in the C Shell, provides a convenient way to specify a series of strings or numbers. Although brace expansion is frequently used to specify filenames, the mechanism can be used to generate arbitrary strings; the shell does not attempt to match the brace notation with the names of existing files. Brace expansion is turned on in interactive and noninteractive shells by default; you can turn it off using **set +o braceexpand**. The shell also uses braces to isolate variable names (page 314).

The following example illustrates how brace expansion works. The ls command does not display any output because there are no files in the working directory. The echo builtin displays the strings the shell generates using brace expansion.

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

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

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

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

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

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

The **–F** option causes ls to display a slash (**/**) after a directory and an asterisk (*****) after an executable file. If you tried to use an ambiguous file reference instead of braces to specify the directories, the result would be different (and not what you wanted):

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

An ambiguous file reference matches the names of existing files. In the preceding example, because it found no filenames matching **vrs[A–E]**, bash passed the ambiguous file reference to mkdir, which created a directory with that name. Brackets in ambiguous file references are discussed on page 155.

Sequence expression

Under newer versions of bash, brace expansion can include a sequence expression to generate a sequence of characters. It can generate a sequential series of numbers or letters using the following syntax:

{n1..n2[..incr]}

where ***n1*** and ***n2*** are numbers or single letters and ***incr*** is a number. This syntax works on bash version 4.0+; give the command **echo $BASH_VERSION** to see which version you are using. The ***incr*** does not work under macOS. When you specify invalid arguments, bash copies the arguments to standard output. Following are some examples:

```
$ echo {4..8}
4 5 6 7 8
$ echo {8..16..2}
8 10 12 14 16
$ echo {a..m..3}
a d g j m
$ echo {a..m..b}
{a..m..b}
$ echo {2..m}
{2..m}
```

See page 500 for a way to use variables to specify the values used by a sequence expression. Page 444 shows an example in which a sequence expression is used to specify step values in a **for...in** loop.

seq

Older versions of bash do not support sequence expressions. Although you can use the seq utility to perform a similar function, seq does not work with letters and displays an error when given invalid arguments. The seq utility uses the following syntax:

*seq **n1** [incr] **n2***

The **–s** option causes seq to use the specified character to separate its output. Following are some examples:

```
$ seq 4 8
4
5
6
7
8

$ seq -s\  8 2 16
8 10 12 14 16

$ seq a d
seq: invalid floating point argument: a
Try 'seq --help' for more information.
```

Tilde Expansion

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

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

```
$ echo $HOME
/home/max
$ echo ~
/home/max
$ echo ~/letter
/home/max/letter
$ cp ~/letter .
```

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

```
$ echo ~zach
/home/zach
$ echo ~root
/root
$ echo ~xx
~xx
```

Tildes are also used in directory stack manipulation (page 307). In addition, **~+** is a synonym for **PWD** (the name of the working directory), and **~–** is a synonym for **OLDPWD** (the name of the previous working directory).

Parameter and Variable Expansion

On a command line, a dollar sign ($) that is not followed by an open parenthesis introduces parameter or variable expansion. *Parameters* include both command-line, or positional, parameters (page 470) and special parameters (page 475). *Variables* include both user-created variables (page 312) and keyword variables (page 317). The bash man and info pages do not make this distinction.

The shell does not expand parameters and variables that are enclosed within single quotation marks and those in which the leading dollar sign is escaped (i.e., preceded with a backslash). The shell does expand parameters and variables enclosed within double quotation marks.

ARITHMETIC EXPANSION

The shell performs *arithmetic expansion* by evaluating an arithmetic expression and replacing it with the result. See page 398 for information on arithmetic expansion under tcsh. Under bash the syntax for arithmetic expansion is

$((expression))

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

The rules for forming ***expression*** are the same as those found in the C programming language; all standard C arithmetic operators are available (see Table 10-8 on page 508). Arithmetic in bash is done using integers. Unless you use variables of type integer (page 316) or actual integers, however, the shell must convert string-valued variables to integers for the purpose of the arithmetic evaluation.

You do not need to precede variable names within ***expression*** with a dollar sign (**$**). In the following example, after read (page 489) assigns the user's response to **age**, an arithmetic expression determines how many years are left until age 100:

```
$ cat age_check
#!/bin/bash
read -p "How old are you? " age
echo "Wow, in $((100-age)) years, you'll be 100!"

$ ./age_check
How old are you? 55
Wow, in 45 years, you'll be 100!
```

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

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

The next example uses wc, cut, arithmetic expansion, and command substitution (page 371) to estimate the number of pages required to print the contents of the file **letter.txt**. The output of the wc (word count) utility (page 1027) used with the **–l** option is the number of lines in the file, in columns (character positions) 1 through 4, followed by a SPACE and the name of the file (the first command following). The cut utility (page 784) with the **–c1–4** option extracts the first four columns.

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

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

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

The preceding example sets up a pipeline that sends standard output from wc to standard input of cut. Because of command substitution, the output of both commands replaces the commands between the $(and the matching) on the command line. Arithmetic expansion then divides this number by 66, the number of lines on a page. A 1 is added because integer division discards remainders.

Fewer dollar signs ($)

tip When you specify variables within **$((** and **))**, the dollar signs that precede individual variable references are optional. This format also allows you to include whitespace around operators, making expressions easier to read.

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

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

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

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

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

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

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

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

You can supply let with multiple arguments on a single command line:

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

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

Command Substitution

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

$(command)

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

`command`

The shell executes ***command*** within a subshell and replaces ***command,*** along with the surrounding punctuation, with standard output of ***command.*** Standard error of ***command*** is not affected.

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

```
$ echo $(pwd)
/home/max
```

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

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

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

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

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

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

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

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

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

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

Try giving this command after giving a **set –x** command (page 442) to see how bash expands it. If there is no **README** file, the command displays the output of **ls –l**.

For additional scripts that use command substitution, see pages 439, 458, and 498.

$((versus $(

tip The symbols **$((** constitute a single token. They introduce an arithmetic expression, not a command substitution. Thus, if you want to use a parenthesized subshell (page 302) within **$()**, you must put a SPACE between the **$(** and the following **(**.

Word Splitting

The results of parameter and variable expansion, command substitution, and arithmetic expansion are candidates for word splitting. Using each character of **IFS** (page 321) as a possible delimiter, bash splits these candidates into words or tokens. If **IFS** is unset, bash uses its default value (SPACE-TAB-NEWLINE). If **IFS** is null, bash does not split words.

Pathname Expansion

Pathname expansion (page 152), also called *filename generation* or *globbing,* is the process of interpreting ambiguous file references and substituting the appropriate list of filenames. Unless **noglob** (page 362) is set, the shell performs this function when it encounters an ambiguous file reference—a token containing any of the unquoted characters *, ?, [, or]. If bash cannot locate any files that match the specified pattern, the token with the ambiguous file reference remains unchanged. The shell does not delete the token or replace it with a null string but rather passes it to the program as is (except see **nullglob** on page 363). The TC Shell generates an error message.

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

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

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

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

A period that either starts a pathname or follows a slash (/) in a pathname must be matched explicitly unless you have set **dotglob** (page 361). The option **nocaseglob** (page 362) causes ambiguous file references to match filenames without regard to case.

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

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

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

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

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

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

Process Substitution

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

The following example uses sort (page 969) with the **–m** (merge, which works correctly only if the input files are already sorted) option to combine two word lists into a single list. Each word list is generated by a pipe that extracts words matching a pattern from a file and sorts the words in that list.

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

Quote Removal

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

Chapter Summary

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

Running a shell script

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

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

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

Job control

A job is another name for a process running a pipeline (which can be a simple command). You can bring a job running in the background into the foreground using the fg builtin. You can put a foreground job into the background using the bg builtin, provided you first suspend the job by pressing the suspend key (typically CONTROL-Z). Use the jobs builtin to display the list of jobs that are running in the background or are suspended.

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

Locale Locale specifies the way locale-aware programs display certain kinds of data, such as times and dates, money and other numeric values, telephone numbers, and measurements. It can also specify collating sequence and printer paper size.

Process Each process is the execution of a single command and has a unique identification (PID) number. When you give the shell a command, it forks a new (child) process to execute the command (unless the command is built into the shell). While the child process is running, the shell is in a state called sleep. By ending a command line with an ampersand (&), you can run a child process in the background and bypass the sleep state so the shell prompt returns immediately after you press RETURN. Each command in a shell script forks a separate process, each of which might in turn fork other processes. When a process terminates, it returns its exit status to its parent process. An exit status of zero signifies success; a nonzero value signifies failure.

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

Command-line editors When using an interactive Bourne Again Shell, you can edit a command line and commands from the history list, using either of the Bourne Again Shell's command-line editors (vim or emacs). When you use the vim command-line editor, you start in Input mode, unlike with the stand-alone version of vim. You can switch between Command and Input modes. The emacs editor is modeless and distinguishes commands from editor input by recognizing control characters as commands.

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

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

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

Command-line expansion When it processes a command line, the Bourne Again Shell replaces some words with expanded text. Most types of command-line expansion are invoked by the appearance of a special character within a word (for example, the leading dollar sign that denotes a variable). Table 8-6 on page 325 lists these special characters. The expansions take place in a specific order. Following the history and alias expansions, the common expansions are parameter and variable expansion, command substitution, and pathname expansion. Surrounding a word with double quotation marks suppresses all types of expansion except parameter and variable expansion. Single quotation marks suppress all types of expansion, as does quoting (escaping) a special character by preceding it with a backslash.

EXERCISES

1. Explain the following unexpected result:

```
$ whereis date
date: /bin/date ...
$ echo $PATH
.:/usr/local/bin:/usr/bin:/bin
$ cat > date
echo "This is my own version of date."
$ ./date
Sun May 21 11:45:49 PDT 2017
```

2. What are two ways you can execute a shell script when you do not have execute permission for the file containing the script? Can you execute a shell script if you do not have read permission for the file containing the script?

3. What is the purpose of the **PATH** variable?

   a. Set the **PATH** variable and place it in the environment so it causes the shell to search the following directories in order:

      - **/usr/local/bin**
      - **/usr/bin**
      - **/bin**
      - **/usr/kerberos/bin**
      - The **bin** directory in your home directory
      - The working directory

   b. If there is an executable file named **doit** in **/usr/bin** and another file with the same name in your **~/bin** directory, which one will be executed?

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

   d. Which command can you use to add the directory **/usr/games** to the end of the list of directories in **PATH**?

4. Assume you have made the following assignment:

```
$ person=zach
```

Give the output of each of the following commands.

a. **echo $person**

b. **echo '$person'**

c. **echo "$person"**

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

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

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

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

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

6. Assume the **/home/zach/grants/biblios** and **/home/zach/biblios** directories exist. Specify Zach's working directory after he executes each sequence of commands. Explain what happens in each case.

a.
```
$ pwd
/home/zach/grants
$ CDPATH=$(pwd)
$ cd
$ cd biblios
```

b.
```
$ pwd
/home/zach/grants
$ CDPATH=$(pwd)
$ cd $HOME/biblios
```

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

8. Enter the following command:

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

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

Advanced Exercises

9. Write a sequence of commands or a script that demonstrates variable expansion occurs before pathname expansion.

10. Write a shell script that outputs the name of the shell executing it.

11. Explain the behavior of the following shell script:

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

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

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

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

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

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

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

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

```
$ dirname simple
.
```

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

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

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

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

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

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

9

The TC Shell (tcsh)

In This Chapter

Objectives

After reading this chapter you should be able to:

- Identify tcsh startup files
- Explain the function of the **history**, **histfile**, and **savehist** variables
- Set up an alias that uses a command-line argument
- Redirect standard error and standard output of a script to two different files
- Set up and use filename, command, and variable completion
- Correct command-line spelling errors
- Explain and use the @ builtin to work with numeric variables
- Explain the use of the **noclobber** variable
- Use an if structure to evaluate the status of a file
- Describe eight tcsh builtins

The TC Shell (tcsh) performs the same function as the Bourne Again Shell and other shells: It provides an interface between you and the Linux operating system. The TC Shell is an interactive command interpreter as well as a high-level programming language. Although you use only one shell at any given time, you should be able to switch back and forth comfortably between shells as the need arises. In fact, you might want to run different shells in different windows. Chapters 8 and 10 apply to tcsh as well as to bash, so they provide a good background for this chapter. This chapter explains tcsh features that are not found in bash and those that are implemented differently from their bash counterparts.

The TC Shell is an expanded version of the C Shell (csh), which originated on Berkeley UNIX. The "T" in TC Shell comes from the TENEX and TOPS-20 operating systems, which inspired command completion and other features in the TC Shell. A number of features not found in csh are present in tcsh, including file and username completion, command-line editing, and spelling correction. As with csh, you can customize tcsh to make it more tolerant of mistakes and easier to use. By setting the proper shell variables, you can have tcsh warn you when you appear to be accidentally logging out or overwriting a file. Many popular features of the original C Shell are now shared by bash and tcsh.

Assignment statement

Although some of the functionality of tcsh is present in bash, differences arise in the syntax of some commands. For example, the tcsh assignment statement has the following syntax:

set ***variable*** *=* ***value***

Having SPACEs on either side of the equal sign, although illegal in bash, is allowed (but not mandatory) in tcsh. By convention shell variables in tcsh are generally named with lowercase letters, not uppercase (you can use either). If you reference an undeclared variable (one that has had no value assigned to it), tcsh generates an error message, whereas by default bash does not. Finally, the default tcsh prompt is a greater than sign (>), but it is frequently set to a single $ character followed by a SPACE. The examples in this chapter use a prompt of **tcsh $** to avoid confusion with the bash prompt.

Do not use tcsh as a programming language

tip

If you have used UNIX and are comfortable with the C or TC Shell, you might want to use tcsh as your login shell. However, you might find that the TC Shell is not as good a programming language as bash. If you are going to learn only one shell programming language, learn bash. The Bourne Again Shell and dash (page 287), which is a subset of bash, are used throughout Linux to program many system administration scripts.

Shell Scripts

The TC Shell can execute files containing tcsh commands, just as the Bourne Again Shell can execute files containing bash commands. Although the concepts of writing and executing scripts in the two shells are similar, the methods of declaring and assigning values to variables and the syntax of control structures are different.

You can run bash and tcsh scripts while using any one of the shells as a command interpreter. Various methods exist for selecting the shell that runs a script. Refer to "#! Specifies a Shell" on page 297 for more information.

If the first character of a shell script is a pound sign (#) and the following character is *not* an exclamation point (!), the TC Shell executes the script under tcsh. If the first character is anything other than #, tcsh calls the sh link to dash or bash to execute the script.

echo: getting rid of the RETURN

tip The tcsh echo builtin accepts either a **–n** option or a trailing **\c** to get rid of the RETURN that echo normally displays at the end of a line. The bash echo builtin accepts only the **–n** option (refer to "read: Accepts User Input" on page 489).

Shell game

tip When you are working with an interactive TC Shell, if you run a script in which **#** is *not* the first character of the script and you call the script *directly* (without preceding its name with tcsh), tcsh calls the sh link to dash or bash to run the script. The following script was written to be run under tcsh but, when called from a tcsh command line, is executed by bash. The set builtin (page 472) works differently under bash and tcsh. As a result the following example (from page 401) issues a prompt but does not wait for you to respond:

```
tcsh $ cat user_in
echo -n "Enter input: "
set input_line = "$<"
echo $input_line

tcsh $ user_in
Enter input:
```

Although in each case the examples are run from a tcsh command line, the following one calls tcsh explicitly so that tcsh executes the script and it runs correctly:

```
tcsh $ tcsh user_in
Enter input: here is some input
here is some input
```

Entering and Leaving the TC Shell

chsh You can execute tcsh by giving the command **tcsh**. If you are not sure which shell you are using, use the ps utility to find out. It shows whether you are running tcsh, bash, sh (linked to bash), or possibly another shell. The finger command followed by your username displays the name of your login shell, which is stored in the **/etc/passwd** file. (macOS uses Open Directory [page 1068] in place of this file.) If you want to use tcsh as a matter of course, you can use the chsh (change shell) utility to change your login shell:

```
bash $ chsh
Changing shell for sam.
Password:
New shell [/bin/bash]: /bin/tcsh
```

```
Shell changed.
bash $
```

The shell you specify will remain in effect for your next login and all subsequent logins until you specify a different login shell. The **/etc/passwd** file stores the name of your login shell.

You can leave tcsh in several ways. The approach you choose depends on two factors: whether the shell variable **ignoreeof** is set and whether you are using the shell that you logged in on (your login shell) or another shell that you created after you logged in. If you are not sure how to exit from tcsh, press CONTROL-D on a line by itself with no leading SPACEs, just as you would to terminate standard input to a program. You will either exit or receive instructions on how to exit. If you have not set **ignoreeof** (page 407) and it has not been set for you in a startup file, you can exit from any shell by using CONTROL-D (the same procedure you use to exit from the Bourne Again Shell).

When **ignoreeof** is set, CONTROL-D does not work. The **ignoreeof** variable causes the shell to display a message telling you how to exit. You can always exit from tcsh by giving an **exit** command. A **logout** command allows you to exit from your login shell only.

Startup Files

When you log in on the TC Shell, it automatically executes various startup files. These files are normally executed in the order described in this section, but you can compile tcsh so that it uses a different order. You must have read access to a startup file to execute the commands in it. See page 288 for information on bash startup files and page 1076 for information on startup files under macOS.

/etc/csh.cshrc and /etc/csh.login The shell first executes the commands in **/etc/csh.cshrc** and **/etc/csh.login**. A user working with **root** privileges can set up these files to establish systemwide default characteristics for tcsh users. They contain systemwide configuration information, such as the default **path**, the location to check for mail, and so on.

.tcshrc and .cshrc Next, the shell looks for **~/.tcshrc** or, if it does not exist, **~/.cshrc** (**~/** is shorthand for your home directory; page 91). You can use these files to establish variables and parameters that are specific to your shell. Each time you create a new shell, tcsh reinitializes these variables for the new shell. The following **.tcshrc** file sets several shell variables; establishes two aliases (page 387); and adds two directories to **path**, one at the beginning of the list and one at the end:

```
tcsh $ cat ~/.tcshrc
set noclobber
set dunique
set ignoreeof
set history=256
set path = (~/bin $path /usr/games)
alias h history
alias ll ls -l
```

.history Login shells rebuild the history list from the contents of **~/.history**. If the **histfile** variable exists, tcsh uses the file that **histfile** points to in place of **.history**.

.login Login shells read and execute the commands in **~/.login**. This file contains commands that you want to execute once, at the beginning of each session. You can use setenv (page 396) to declare environment (global) variables here. You can also declare the type of terminal you are using and set some terminal characteristics in your **.login** file.

```
tcsh $ cat ~/.login
setenv history 200
setenv mail /var/spool/mail/$user
if ( -z $DISPLAY ) then
        setenv TERM vt100
    else
        setenv TERM xterm
endif
stty erase '^h' kill '^u' -lcase tab3
date '+Login on %A %B %d at %I:%M %p'
```

The preceding **.login** file establishes the type of terminal you are using by setting the **TERM** variable (the **if** statement [page 409] determines whether you are using a graphical interface and therefore which value should be assigned to **TERM**). It then runs stty (page 987) to set terminal characteristics and date (page 787) to display the time you logged in.

/etc/csh.logout and .logout The TC Shell runs the **/etc/csh.logout** and **~/.logout** files, in that order, when you exit from a login shell. The following sample **.logout** file uses date to display the time you logged out. The sleep command ensures that echo has time to display the message before the system logs you out. The delay might be useful for dial-up lines that take some time to display the message.

```
tcsh $ cat ~/.logout
date '+Logout on %A %B %d at %I:%M %p'
sleep 5
```

Features Common to the Bourne Again and TC Shells

Most of the features common to both bash and tcsh are derived from the original C Shell:

- Command-line expansion (also called substitution; next page)
- History (next page)
- Aliases (page 387)
- Job control (page 388)
- Filename substitution (page 388)
- Directory stack manipulation (page 389)
- Command substitution (page 389)

The chapters on bash discuss these features in detail. This section focuses on the differences between the bash and tcsh implementations.

Command-Line Expansion (Substitution)

Refer to “Processing the Command Line” on page 364 for an introduction to command-line expansion in the Bourne Again Shell. The tcsh man page uses the term *substitution* instead of *expansion;* the latter is used by bash. The TC Shell scans each token on a command line for possible expansion in the following order:

1. History substitution (below)
2. Alias substitution (page 387)
3. Variable substitution (page 396)
4. Command substitution (page 389)
5. Filename substitution (page 388)
6. Directory stack substitution (page 389)

History

The TC Shell assigns a sequential *event number* to each command line. You can display this event number as part of the tcsh prompt (refer to “prompt” on page 403). Examples in this section show numbered prompts when they help illustrate the behavior of a command.

The history Builtin

As in bash, the tcsh history builtin displays the events in your history list. The list of events is ordered with the oldest events at the top. The last event in the history list is the **history** command that displayed the list. In the following history list, which is limited to ten lines by the argument of **10** to the **history** command, command 23 modifies the tcsh prompt to display the history event number. The time each command was executed appears to the right of the event number.

```
32 $ history 10
    23  23:59   set prompt = "! $ "
    24  23:59   ls -l
    25  23:59   cat temp
    26  0:00    rm temp
    27  0:00    vim memo
    28  0:00    lpr memo
    29  0:00    vim memo
    30  0:00    lpr memo
    31  0:00    rm memo
    32  0:00    history
```

History Expansion

The same event and word designators work in both shells. For example, **!!** refers to the previous event in tcsh, just as it does in bash. The command **!328** executes event number 328; **!?txt?** executes the most recent event containing the string **txt**. For more information refer to "Using an Exclamation Point (!) to Reference Events" on page 341. Table 9-1 lists the few tcsh word modifiers not found in bash.

Table 9-1 Word modifiers

Modifier	Function
u	Converts the first lowercase letter into uppercase
l	Converts the first uppercase letter into lowercase
a	Applies the next modifier globally within a single word

You can use more than one word modifier in a command. For instance, the **a** modifier, when used in combination with the **u** or **l** modifier, enables you to change the case of an entire word.

```
tcsh $ echo $VERSION
VERSION: Undefined variable.
tcsh $ echo !!:1:al
echo $version
tcsh 6.17.00 (Astron) 2009-07-10 (i386-intel-linux) options wide,nls, ...
```

In addition to using event designators to access the history list, you can use the command-line editor to access, modify, and execute previous commands (page 393).

Variables

The variables you set to control the history list in tcsh are different from those used in bash. Whereas bash uses **HISTSIZE** and **HISTFILESIZE** to determine the number of events that are preserved during and between sessions, respectively, tcsh uses **history** and **savehist** (Table 9-2) for these purposes.

Table 9-2 History variables

Variable	Default	Function
history	100 events	Maximum number of events saved during a session
histfile	**~/.history**	Location of the history file
savehist	not set	Maximum number of events saved between sessions

history and **savehist** When you exit from a tcsh shell, the most recently executed commands are saved in your **~/.history** file. The next time you start the shell, this file initializes the history list. The value of the **savehist** variable determines the number of lines saved in the **.history** file (not necessarily the same as the **history** variable). If **savehist** is not set, tcsh does not save history information between sessions. The **history** and **savehist** variables must be shell variables (i.e., declared using set, not setenv). The **history** variable holds the number of events remembered during a session and the **savehist** variable holds the number remembered between sessions. See Table 9-2.

If you set the value of **history** too high, it can use too much memory. If it is unset or set to zero, the shell does not save any commands. To establish a history list of the

500 most recent events, give the following command manually or place it in your **~/.tcshrc** startup file:

```
tcsh $ set history = 500
```

The following command causes tcsh to save the 200 most recent events across login sessions:

```
tcsh $ set savehist = 200
```

You can combine these two assignments into a single command:

```
tcsh $ set history=500 savehist=200
```

After you set **savehist**, you can log out and log in again; the 200 most recent events from the previous login sessions will appear in your history list after you log back in. Set **savehist** in your **~/.tcshrc** file if you want to maintain your event list from login to login.

histlit If you set the variable **histlit** (history literal), history displays the commands in the history list exactly as they were typed in, without any shell interpretation. The following example shows the effect of this variable (compare the lines numbered 32):

```
tcsh $ cat /etc/csh.cshrc
...
tcsh $ cp !!:1 ~
cp /etc/csh.cshrc ~
tcsh $ set histlit
tcsh $ history
...
    31  9:35    cat /etc/csh.cshrc
    32  9:35    cp !!:1 ~
    33  9:35    set histlit
    34  9:35    history
tcsh $ unset histlit
tcsh $ history
...
    31  9:35    cat /etc/csh.cshrc
    32  9:35    cp /etc/csh.cshrc ~
    33  9:35    set histlit
    34  9:35    history
    35  9:35    unset histlit
    36  9:36    history
```

optional The bash and tcsh Shells expand history event designators differently. If you give the command **!250w**, bash replaces it with command number 250 with a **w** character appended to it. In contrast, tcsh looks back through your history list for an event that begins with the string **250w** to execute. The reason for the difference: bash interprets the first three characters of **250w** as the number of a command, whereas tcsh interprets those characters as part of the search string **250w**. (If the 250 stands alone, tcsh treats it as a command number.)

If you want to append **w** to command number 250, you can insulate the event number from the **w** by surrounding it with braces:

```
!{250}w
```

Aliases

The alias/unalias feature in tcsh closely resembles its counterpart in bash (page 352). However, the alias builtin has a slightly different syntax:

*alias **name value***

The following command creates an alias for **ls**:

```
tcsh $ alias ls "ls -lF"
```

The tcsh alias allows you to substitute command-line arguments, whereas bash does not:

```
tcsh $ alias nam "echo Hello, \!^ is my name"
tcsh $ nam Sam
Hello, Sam is my name
```

The string **\!*** within an alias expands to all command-line arguments:

```
tcsh $ alias sortprint "sort \!* | lpr"
```

The next alias displays its second argument:

```
tcsh $ alias n2 "echo \!:2"
```

To display a list of current aliases, give the command **alias**. To display the alias for a particular name, give the command **alias** followed by that name.

Special Aliases

Some alias names, called *special aliases,* have special meaning to tcsh. If you define an alias that uses one of these names, tcsh executes it automatically as explained in Table 9-3. Initially, all special aliases are undefined. The following command sets the **cwdcmd** alias so it displays the name of the working directory when you change to a new working directory. The single quotation marks are critical in this example; see page 353.

```
tcsh $ alias cwdcmd 'echo Working directory is now `pwd`'
tcsh $ cd /etc
Working directory is now /etc
tcsh $
```

Table 9-3 Special aliases

Alias	When executed
beepcmd	Whenever the shell would normally ring the terminal bell. Gives you a way to have other visual or audio effects take place at those times.
cwdcmd	Whenever you change to another working directory.
periodic	Periodically, as determined by the number of minutes in the **tperiod** variable. If **tperiod** is unset or has the value 0, **periodic** has no meaning.

Table 9-3 Special aliases (continued)

Alias	When executed
precmd	Just before the shell displays a prompt.
shell	Specifies the absolute pathname of the shell that will run scripts that do not start with **#!** (page 297).

History Substitution in Aliases

You can substitute command-line arguments by using the history mechanism, where a single exclamation point represents the command line containing the alias. Modifiers are the same as those used by history (page 341). In the following example, the exclamation points are quoted so the shell does not interpret them when building the aliases:

```
21 $ alias last echo \!:$
22 $ last this is just a test
test
23 $ alias fn2 echo \!:2:t
24 $ fn2 /home/sam/test /home/zach/temp /home/barbara/new
temp
```

Event 21 defines for **last** an alias that displays the last argument. Event 23 defines for **fn2** an alias that displays the simple filename, or tail, of the second argument on the command line.

Job Control

Job control is similar in both bash (page 304) and tcsh. You can move commands between the foreground and the background, suspend jobs temporarily, and display a list of current jobs. The **%** character references a job when it is followed by a job number or a string prefix that uniquely identifies the job. You will see a minor difference when you run a multiple-process command line in the background from each shell. Whereas bash displays only the PID number of the last background process in each job, tcsh displays the numbers for all processes belonging to a job. The example from page 304 looks like this under tcsh:

```
tcsh $ find . -print | sort | lpr & grep -l max /tmp/* > maxfiles &
[1] 18839  18840  18841
[2] 18876
```

Filename Substitution

The TC Shell expands the characters *, ?, and [ ] in a pathname just as bash does (page 152). The * matches any string of zero or more characters, ? matches any single character, and [ ] defines a character class that matches single characters appearing between the brackets.

The TC Shell expands command-line arguments that start with a tilde (~) into filenames in much the same way that bash does (page 391), with the ~ standing for the user's home directory or the home directory of the user whose name follows the tilde. The bash special expansions ~+ and ~– are not available in tcsh.

Brace expansion (page 366) is available in tcsh. Like tilde expansion, it is regarded as an aspect of filename substitution even though brace expansion can generate strings that are not the names of existing files.

globbing In tcsh and its predecessor csh, the process of using patterns to match filenames is referred to as *globbing* and the pattern itself is called a *globbing pattern*. If tcsh is unable to identify one or more files that match a globbing pattern, it reports an error (unless the pattern contains a brace). Setting the shell variable **noglob** suppresses filename substitution, including both tilde and brace interpretation.

Manipulating the Directory Stack

Directory stack manipulation in tcsh does not differ much from that in bash (page 307). The dirs builtin displays the contents of the stack, and the pushd and popd builtins push directories onto and pop directories off of the stack.

Command Substitution

The **$(...)** syntax for command substitution is *not* available in tcsh. In its place you must use the original **'...'** syntax. Otherwise, the implementation in bash and tcsh is identical. Refer to page 371 for more information on command substitution.

Redirecting Standard Error

Both bash and tcsh use a greater than symbol (**>**) to redirect standard output, but tcsh does *not* use the bash notation **2>** to redirect standard error. Under tcsh you use a greater than symbol followed by an ampersand (**>&**) to combine and redirect standard output and standard error. Although you can use this notation under bash, few people do. The following examples, like the bash examples on page 292, reference file **x**, which does not exist, and file **y**, which contains a single line:

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

With an argument of **y** in the preceding example, cat sends a string to standard output. An argument of **x** causes cat to send an error message to standard error.

Unlike bash, tcsh does not provide a simple way to redirect standard error separately from standard output. A work-around frequently provides a reasonable solution. The following example runs cat with arguments of **x** and **y** in a subshell (the parentheses ensure that the command within them runs in a subshell; page 302). Also within the subshell, a **>** redirects standard output to the file **outfile**. Output sent to standard error is not touched by the subshell but rather is sent to the parent shell, where both it and standard output are sent to **errfile**. Because standard output has already been redirected, **errfile** contains only output sent to standard error.

```
tcsh $ (cat x y > outfile) >& errfile
tcsh $ cat outfile
This is y.
tcsh $ cat errfile
cat: x: No such file or directory
```

It can be useful to combine and redirect output when you want to execute a command that runs slowly in the background and do not want its output cluttering up the screen. For example, because the find utility (page 822) can take a long time to complete, it might be a good idea to run it in the background. The next command finds in the filesystem hierarchy all files that contain the string **biblio** in their name. This command runs in the background and sends its output to the **findout** file. Because the find utility sends to standard error a report of directories that you do not have permission to search, the **findout** file contains a record of any files that are found as well as a record of the directories that could not be searched.

```
tcsh $ find / -name "*biblio*" -print >& findout &
```

In this example, if you did not combine standard error with standard output and redirected only standard output, the error messages would appear on the screen and **findout** would list only files that were found.

While a command that has its output redirected to a file is running in the background, you can look at the output by using tail (page 992) with the **–f** option. The **–f** option causes tail to display new lines as they are written to the file:

```
tcsh $ tail -f findout
```

To terminate the tail command, press the interrupt key (usually CONTROL-C).

Working with the Command Line

This section covers word completion, editing the command line, and correcting spelling.

Word Completion

The TC Shell completes filenames, commands, and variable names on the command line when you prompt it to do so. The generic term used to refer to all these features under tcsh is *word completion.*

Filename Completion

The TC Shell can complete a filename after you specify a unique prefix. Filename completion is similar to filename generation, but the goal of filename completion is to select a single file. Together these capabilities make it practical to use long, descriptive filenames.

To use filename completion when you are entering a filename on the command line, type enough of the name to identify the file uniquely and press TAB; tcsh fills in the name and adds a SPACE, leaving the cursor so you can enter additional arguments or press RETURN. In the following example, the user types the command **cat trig1A** and presses TAB; the system fills in the rest of the filename that begins with **trig1A**:

```
tcsh $ cat trig1A ⇨ TAB ⇨ cat trig1A.302488 ■
```

If two or more filenames match the prefix that you have typed, tcsh cannot complete the filename without obtaining more information. The shell maximizes the length of the prefix by adding characters, if possible, and then beeps to signify that additional input is needed to resolve the ambiguity:

```
tcsh $ ls h*
help.hist  help.trig01  help.txt
tcsh $ cat h ⇨ TAB ⇨ cat help. (beep)
```

You can fill in enough characters to resolve the ambiguity and then press the TAB key again. Alternatively, you can press CONTROL-D to cause tcsh to display a list of matching filenames:

```
tcsh $ cat help. ⇨ CONTROL-D
help.hist    help.trig01  help.txt
tcsh $ cat help.■
```

After displaying the filenames, tcsh redraws the command line so you can disambiguate the filename (and press TAB again) or finish typing the filename manually.

Tilde Completion

The TC Shell parses a tilde (~) appearing as the first character of a word and attempts to expand it to a username when you enter a TAB:

```
tcsh $ cd ~za ⇨ TAB ⇨ cd ~zach/■ ⇨ RETURN
tcsh $ pwd
/home/zach
```

By appending a slash (/), tcsh indicates that the completed word is a directory. The slash also makes it easy to continue specifying a pathname.

Command and Variable Completion

You can use the same mechanism you use to list and complete filenames with command and variable names. When you specify a simple filename, the shell uses the variable **path** to attempt to complete a command name. The choices tcsh lists might be located in different directories.

```
tcsh $ up ⇨ TAB (beep) ⇨ CONTROL-D
up2date              updatedb             uptime
up2date-config       update-mime-database
up2date-nox          updmap
tcsh $ up ⇨ t ⇨ TAB ⇨ uptime ■ ⇨ RETURN
9:59am up 31 days, 15:11, 7 users, load average: 0.03, 0.02, 0.00
```

If you set the **autolist** variable as in the following example, the shell lists choices automatically when you invoke completion by pressing TAB. You do not have to press CONTROL-D.

```
tcsh $ set autolist
tcsh $ up ⇨ TAB (beep)
up2date              updatedb             uptime
up2date-config       update-mime-database
up2date-nox          updmap
tcsh $ up ⇨ t ⇨ TAB ⇨ uptime ■ ⇨ RETURN
10:01am up 31 days, 15:14, 7 users, load average: 0.20, 0.06, 0.02
```

If you set **autolist** to **ambiguous**, the shell lists the choices when you press TAB *only* if the word you enter is the longest prefix of a set of commands. Otherwise, pressing TAB causes the shell to add one or more characters to the word until it is the longest prefix; pressing TAB again then lists the choices:

```
tcsh $ set autolist=ambiguous
tcsh $ echo $h ⇨ TAB (beep)
histfile history home
tcsh $ echo $h■ ⇨ i ⇨ TAB ⇨ echo $hist■ ⇨ TAB
histfile history
tcsh $ echo $hist■ ⇨ o ⇨ TAB ⇨ echo $history ■ ⇨ RETURN
1000
```

The shell must rely on the context of the word within the input line to determine whether it is a filename, a username, a command, or a variable name. The first word on an input line is assumed to be a command name; if a word begins with the special character **$**, it is viewed as a variable name; and so on. In the following example, the second which command does not work properly: The context of the word **up** makes it look like the beginning of a filename rather than the beginning of a command. The TC Shell supplies which with an argument of **updates** (a nonexecutable file) and which displays an error message:

```
tcsh $ ls up*
updates
tcsh $ which updatedb ups uptime
/usr/bin/updatedb
/usr/local/bin/ups
/usr/bin/uptime

tcsh $ which up ⇨ TAB ⇨ which updates
updates: Command not found.
```

Editing the Command Line

bindkey The tcsh command-line editing feature is similar to that available under bash. You can use either emacs mode commands (default) or vi(m) mode commands. Change to vi(m) mode commands by giving the command **bindkey –v** and to emacs mode commands by giving the command **bindkey –e**. The ARROW keys are bound to the obvious motion commands in both modes, so you can move back and forth (up and down) through the history list as well as left and right on the current command line.

Without an argument, the bindkey builtin displays the current mappings between editor commands and the key sequences you can enter at the keyboard:

```
tcsh $ bindkey
Standard key bindings
"^@"          ->  set-mark-command
"^A"          ->  beginning-of-line
"^B"          ->  backward-char
"^C"          ->  tty-sigintr
"^D"          ->  delete-char-or-list-or-eof
...
Multi-character bindings
"^[[A"        -> up-history
"^[[B"        -> down-history
"^[[C"        -> forward-char
"^[[D"        -> backward-char
"^[[H"        -> beginning-of-line
...
Arrow key bindings
down          -> down-history
up            -> up-history
left          -> backward-char
right         -> forward-char
home          -> beginning-of-line
end           -> end-of-line
```

The **^** indicates a CONTROL character (**^B** = CONTROL-B). The **^[** indicates a META or ALT character; in this case you press and hold the META or ALT key while you press the key for the next character. If this substitution does not work or if the keyboard you are using does not have a META or ALT key, press and release the ESCAPE key and then press the key for the next character. For **^[[F** you would press META-[or ALT-[followed by the **F** key or else ESCAPE **[F**. The **down/up/left/right** indicate ARROW keys, and **home/end** indicate the HOME and END keys on the numeric keypad. See page 231 for more information on the META key.

Under macOS, most keyboards do not have a META or ALT key. See page 1076 for an explanation of how to set up the OPTION key to perform the same functions as the META key on a Macintosh.

The preceding example shows the output from bindkey with the user in emacs mode. Change to vi(m) mode (**bindkey –v**) and give another bindkey command to display the vi(m) key bindings. You can send the output of bindkey through a pipeline to less to make it easier to read.

CORRECTING SPELLING

You can have tcsh attempt to correct the spelling of command names, filenames, and variables (but only using emacs-style key bindings). Spelling correction can take place before and after you press RETURN.

BEFORE YOU PRESS RETURN

For tcsh to correct a word or line before you press RETURN, you must indicate that you want it to do so. The two functions for this purpose are **spell-line** and **spell-word**:

```
$ bindkey | grep spell
"^[$"          -> spell-line
"^[S"          -> spell-word
"^[s"          -> spell-word
```

The output from bindkey shows that **spell-line** is bound to META-$ (ALT-$ or ESCAPE **$**) and **spell-word** is bound to META-S and META-s (ALT-s or ESCAPE **s** and ALT-S or ESCAPE **S**). To correct the spelling of the word to the left of the cursor, press META-s. Pressing META-$ invokes the **spell-line** function, which attempts to correct all words on a command line:

```
tcsh $ ls
bigfile.gz
tcsh $ gunzipp ⇨ META-s ⇨ gunzip bigfele.gz ⇨ META-s ⇨ gunzip bigfile.gz
tcsh $ gunzip bigfele.gz ⇨ META-$ ⇨ gunzip bigfile.gz
tcsh $ ecno $usfr ⇨ META-$ ⇨ echo $user
```

AFTER YOU PRESS RETURN

The variable named **correct** controls what tcsh attempts to correct or complete *after* you press RETURN and before it passes the command line to the command being called. If you do not set **correct**, tcsh will not correct anything:

```
tcsh $ unset correct
tcsh $ ls morning
morning
tcsh $ ecno $usfr morbing
usfr: Undefined variable.
```

The shell reports the error in the variable name and not the command name because it expands variables before it executes the command (page 384). When you give a bad command name without any arguments, the shell reports on the bad command name.

Set **correct** to **cmd** to correct only commands; to **all** to correct commands, variables, and filenames; or to **complete** to complete commands:

```
tcsh $ set correct = cmd
tcsh $ ecno $usfr morbing

CORRECT>echo $usfr morbing (y|n|e|a)? y
usfr: Undefined variable.
tcsh $ set correct = all
tcsh $ echo $usfr morbing

CORRECT>echo $user morning (y|n|e|a)? y
zach morning
```

With **correct** set to **cmd**, tcsh corrects the command name from **ecno** to **echo**. With **correct** set to **all**, tcsh corrects both the command name and the variable. It would also correct a filename if one was present on the command line.

The TC Shell displays a special prompt that lets you enter **y** to accept the modified command line, **n** to reject it, **e** to edit it, or **a** to abort the command. Refer to **prompt3** on page 405 for a discussion of the special prompt used in spelling correction.

In the next example, after setting the **correct** variable the user mistypes the name of the ls command; tcsh then prompts for a correct command name. Because the command that tcsh has offered as a replacement is not ls, the user chooses to edit the command line. The shell leaves the cursor following the command so the user can correct the mistake:

```
tcsh $ set correct=cmd
tcsh $ lx -l ⇨ RETURN (beep)
CORRECT>lex -l (y|n|e|a)? e
tcsh $ lx -l█
```

If you assign the value **complete** to the variable **correct**, tcsh attempts command name completion in the same manner as filename completion (page 391). In the following example, after setting **correct** to **complete** the user enters the command **up**. The shell responds with **Ambiguous command** because several commands start with these two letters but differ in the third letter. The shell then redisplays the command line. The user could press TAB at this point to get a list of commands that start with **up** but decides to enter **t** and press RETURN. The shell completes the command because these three letters uniquely identify the uptime utility:

```
tcsh $ set correct = complete
tcsh $ upRETURN
Ambiguous command
tcsh $ up ⇨ tRETURN ⇨ uptime
4:45pm  up 5 days,  9:54,  5 users,  load average: 1.62, 0.83, 0.33
```

Variables

Although tcsh stores variable values as strings, you can work with these variables as numbers. Expressions in tcsh can use arithmetic, logical, and conditional operators. The @ builtin can evaluate integer arithmetic expressions.

This section uses the term *numeric variable* to describe a string variable that contains a number that tcsh uses in arithmetic or logical arithmetic computations. However, no true numeric variables exist in tcsh.

Variable name A tcsh variable name consists of 1 to 20 characters, which can be letters, digits, and underscores (_). The first character cannot be a digit but can be an underscore.

Variable Substitution

Three builtins declare, display, and assign values to variables: set, @, and setenv. The set and setenv builtins both assume nonnumeric string variables. The @ builtin works only with numeric variables. Both set and @ declare shell (local) variables. The setenv builtin declares an environment (global) variable. Using setenv is similar to assigning a value to a variable and then using export in the Bourne Again Shell. See "Environment, Environment Variables, and Inheritance" on page 480 for a discussion of shell and environment variables.

Once the value—or merely the existence—of a variable has been established, tcsh substitutes the value of that variable when the name of the variable, preceded by a dollar sign ($), appears on a command line. If you quote the dollar sign by preceding it with a backslash or enclosing it within single quotation marks, the shell does not perform the substitution. When a variable is within double quotation marks, the substitution occurs even if you quote the dollar sign by preceding it with a backslash.

String Variables

The TC Shell treats string variables similarly to the way the Bourne Again Shell does. The major difference lies in their declaration and assignment: tcsh uses an explicit command, set (or setenv), to declare and/or assign a value to a string variable.

```
tcsh $ set name = fred
tcsh $ echo $name
fred
tcsh $ set
argv    ()
```

```
cwd     /home/zach
home    /home/zach
name    fred
path    (/usr/local/bin /bin /usr/bin /usr/X11R6/bin)
prompt  $
shell   /bin/tcsh
status  0
term    vt100
user    zach
```

The first line in the example declares the variable **name** and assigns the string **fred** to it. Unlike bash, tcsh allows—but does not require—SPACEs around the equal sign. The next line displays the value of **name**. When you give a set command without any arguments, it displays a list of all shell (not environment) variables and their values. When you give a set command with the name of a variable and no value, the command sets the value of the variable to the null string.

You can use the unset builtin to remove a variable:

```
tcsh $ set name
tcsh $ echo $name

tcsh $ unset name
tcsh $ echo $name
name: Undefined variable.
```

setenv The setenv builtin declares an environment variable. When using setenv you must separate the variable name from the string being assigned to it by inserting one or more SPACEs and omitting the equal sign. In the following example, the **tcsh** command creates a subshell, echo shows that the variable and its value are known to the subshell, and **exit** returns to the original shell. Try this example, using set in place of setenv:

```
tcsh $ setenv SRCDIR /usr/local/src
tcsh $ tcsh
tcsh $ echo $SRCDIR
/usr/local/src
tcsh $ exit
```

Without arguments, setenv displays a list of the environment (global) variables—variables that are passed to the shell's child processes. By convention, environment variables are named using uppercase letters.

As with set, giving setenv a variable name without a value sets the value of the variable to a null string. Although you can use unset to remove environment and local variables, unsetenv can remove environment variables only.

ARRAYS OF STRING VARIABLES

An *array* is a collection of strings, each of which is identified by its index (1, 2, 3, and so on). Arrays in tcsh use one-based indexing (i.e., the first element of the array has the subscript 1). Before you can access individual elements of an array, you must declare the entire array by assigning a value to each element of the array. The list of values must be enclosed in parentheses and separated by SPACEs:

```
8 $ set colors = (red green blue orange yellow)
9 $ echo $colors
red green blue orange yellow
10 $ echo $colors[3]
blue
11 $ echo $colors[2-4]
green blue orange
12 $ set shapes = ('' '' '' '' '')
13 $ echo $shapes

14 $ set shapes[4] = square
15 $ echo $shapes[4]
square
```

Event 8 declares the array of string variables named **colors** to have five elements and assigns values to each of them. If you do not know the values of the elements at the time you declare an array, you can declare an array containing the necessary number of null elements (event 12).

You can reference an entire array by preceding its name with a dollar sign (event 9). A number in brackets following a reference to the array refers to an element of the array (events 10, 14, and 15). Two numbers in brackets, separated by a hyphen, refer to two or more adjacent elements of the array (event 11). Refer to "Special Variable Forms" on page 401 for more information on arrays.

NUMERIC VARIABLES

The @ builtin assigns the result of a numeric calculation to a numeric variable (as described under "Variables" on page 396, tcsh has no true numeric variables). You can declare single numeric variables using @, just as you can use set to declare nonnumeric variables. However, if you give it a nonnumeric argument, @ displays an error message. Just as set does, the @ command used without any arguments lists all shell variables.

Many of the expressions that the @ builtin can evaluate and the operators it recognizes are derived from the C programming language. The following syntax shows a declaration or assignment using @ (the SPACE after the @ is required):

@ ***variable-name operator expression***

The ***variable-name*** is the name of the variable you are assigning a value to. The ***operator*** is one of the C assignment operators: **=**, **+=**, **–=**, ***=**, **/=**, or **%=**. (See Table 14-4 on page 641 for a description of these operators.) The ***expression*** is an arithmetic expression that can include most C operators (see the next section). You can use parentheses within the expression for clarity or to change the order of evaluation.

Parentheses must surround parts of the expression that contain any of the following characters: <, >, &, or |.

Do not use $ when assigning a value to a variable

tip As with bash, variables having a value assigned to them (those on the left of the operator) must not be preceded by a dollar sign (**$**) in tcsh. Thus,

```
tcsh $ @ $answer = 5 + 5
```

will yield

```
answer: Undefined variable.
```

or, if **answer** is defined,

```
@: Variable name must begin with a letter.
```

whereas

```
tcsh $ @ answer = 5 + 5
```

assigns the value 10 to the variable **answer**.

EXPRESSIONS

An expression can be composed of constants, variables, and most of the bash operators (page 508). Expressions that involve files rather than numeric variables or strings are described in Table 9-8 on page 409.

Expressions follow these rules:

1. The shell evaluates a missing or null argument as 0.
2. All results are decimal numbers.
3. Except for **!=** and **==**, the operators act on numeric arguments.
4. You must separate each element of an expression from adjacent elements by a SPACE, unless the adjacent element is &, |, <, >, (, or).

Following are some examples that use @:

```
216 $ @ count = 0
217 $ echo $count
0
218 $ @ count = ( 10 + 4 ) / 2
219 $ echo $count
7
220 $ @ result = ( $count < 5 )
221 $ echo $result
0
222 $ @ count += 5
223 $ echo $count
12
224 $ @ count++
225 $ echo $count
13
```

Event 216 declares the variable **count** and assigns it a value of 0. Event 218 shows the result of an arithmetic operation being assigned to a variable. Event 220 uses the @ symbol to assign the result of a logical operation involving a constant and a variable to **result**. The value of the operation is *false* (= 0) because the variable **count** is not less than 5. Event 222 is a compressed form of the following assignment statement:

```
tcsh $ @ count = $count + 5
```

Event 224 uses a postfix operator to increment **count** by 1.

Postincrement and postdecrement operators

You can use the postincrement (**++**) and postdecrement (**––**) operators only in expressions containing a single variable name, as shown in the following example:

```
tcsh $ @ count = 0
tcsh $ @ count++
tcsh $ echo $count
1
tcsh $ @ next = $count++
@: Badly formed number.
```

Unlike in the C programming language and bash, expressions in tcsh cannot use preincrement and predecrement operators.

Arrays of Numeric Variables

You must use the set builtin to declare an array of numeric variables before you can use @ to assign values to the elements of that array. The set builtin can assign any values to the elements of a numeric array, including zeros, other numbers, and null strings.

Assigning a value to an element of a numeric array is similar to assigning a value to a simple numeric variable. The only difference is that you must specify the element, or index, of the array. The syntax is

@ ***variable-name[index] operator expression***

The ***index*** specifies the element of the array that is being addressed. The first element has an index of 1. The ***index*** cannot be an expression but rather must be either a numeric constant or a variable. In the preceding syntax the brackets around ***index*** are part of the syntax and do not indicate that ***index*** is optional. If you specify an ***index*** that is too large for the array you declared with set, tcsh displays **@: Subscript out of range**.

```
226 $ set ages = (0 0 0 0 0)
227 $ @ ages[2] = 15
228 $ @ ages[3] = ($ages[2] + 4)
229 $ echo $ages[3]
19
230 $ echo $ages
0 15 19 0 0
231 $ set index = 3
232 $ echo $ages[$index]
```

```
19
233 $ echo $ages[6]
ages: Subscript out of range.
```

Elements of a numeric array behave as though they were simple numeric variables. Event 226 declares an array with five elements, each having a value of 0. Events 227 and 228 assign values to elements of the array, and event 229 displays the value of one of the elements. Event 230 displays all the elements of the array, event 232 specifies an element by using a variable, and event 233 demonstrates the out-of-range error message.

BRACES

Like bash, tcsh allows you to use braces to distinguish a variable from the surrounding text without the use of a separator:

```
$ set bb=abc
$ echo $bbdef
bbdef: Undefined variable.
$ echo ${bb}def
abcdef
```

SPECIAL VARIABLE FORMS

The special variable with the following syntax has the value of the number of elements in the ***variable-name*** array:

$#variable-name

You can determine whether ***variable-name*** has been set by looking at the value of the variable with the following syntax:

$?variable-name

This variable has a value of 1 if ***variable-name*** is set and 0 otherwise:

```
tcsh $ set days = (mon tues wed thurs fri)
tcsh $ echo $#days
5
tcsh $ echo $?days
1
tcsh $ unset days
tcsh $ echo $?days
0
```

READING USER INPUT

Within a tcsh shell script, you can use the set builtin to read a line from the terminal and assign it to a variable. The following portion of a shell script prompts the user and reads a line of input into the variable **input_line**:

```
echo -n "Enter input: "
set input_line = "$<"
```

The value of the shell variable **$<** is a line from standard input. The quotation marks around **$<** keep the shell from assigning only the first word of the line of input to the variable **input_line**.

tcsh VARIABLES

TC Shell variables can be set by the shell, inherited by the shell from its parent, or set by the user and used by the shell. Some variables take on significant values (for example, the PID number of a background process). Other variables act as switches: *on* if they are declared and *off* if they are not. Many of the shell variables are often set from a startup file (page 382).

tcsh VARIABLES THAT TAKE ON VALUES

argv Contains the command-line arguments (positional parameters) from the command line that invoked the shell. Like all tcsh arrays, this array uses one-based indexing; **argv[1]** contains the first command-line argument. You can abbreviate references to **$argv[*n*]** as **$*n***. The token **argv[*]** references all the arguments together; you can abbreviate it as **$***. Use **$0** to reference the name of the calling program. Refer to "Positional Parameters" on page 470. The Bourne Again Shell does not use the **argv** form, only the abbreviated form. You cannot assign values to the elements of **argv**.

$#argv *or* **$#** Holds the number of elements in the **argv** array. Refer to "Special Variable Forms" on page 401.

autolist Controls command and variable completion (page 392).

autologout Enables tcsh's automatic logout facility, which logs you out if you leave the shell idle for too long. The value of the variable is the number of minutes of inactivity that tcsh waits before logging you out. The default is 60 minutes except when you are running in a graphical environment, in which case this variable is initially unset.

cdpath Affects the operation of cd in the same way as the **CDPATH** variable does in bash (page 323). The **cdpath** variable is assigned an array of absolute pathnames (see **path**, later in this section) and is usually set in the **~/.login** file with a line such as the following:

```
set cdpath = (/home/zach /home/zach/letters)
```

When you call cd with a simple filename, it searches the working directory for a subdirectory with that name. If one is not found, cd searches the directories listed in **cdpath** for the subdirectory.

correct Set to **cmd** for automatic spelling correction of command names, to **all** to correct the entire command line, and to **complete** for automatic completion of command names. This variable works on corrections that are made after you press RETURN. Refer to "After You Press RETURN" on page 394.

cwd The shell sets this variable to the name of the working directory. When you access a directory through a symbolic link (page 115), tcsh sets **cwd** to the name of the symbolic link.

dirstack The shell keeps the stack of directories used with the pushd, popd, and dirs builtins in this variable. For more information refer to "Manipulating the Directory Stack" on page 307.

fignore Holds an array of suffixes that tcsh ignores during filename completion.

gid The shell sets this variable to your group ID.

histfile Holds the full pathname of the file that saves the history list between login sessions (page 385). The defaults is **~/.history**.

history Specifies the size of the history list. Refer to "History" on page 384.

home *or* **HOME** Holds the pathname of the user's home directory. The cd builtin refers to this variable, as does the filename substitution of ~ (page 368).

mail Specifies files and directories, separated by whitespace, to check for mail. The TC Shell checks for new mail every 10 minutes unless the first word of **mail** is a number, in which case that number specifies how often the shell should check in seconds.

owd The shell keeps the name of your previous (old) working directory in this variable, which is equivalent to **~–** in bash.

path *or* **PATH** Holds a list of directories that tcsh searches for executable commands (page 318). If this array is empty or unset, you can execute commands only by giving their pathnames. You can set **path** with a command such as the following:

```
tcsh $ set path = ( /usr/bin /bin /usr/local/bin /usr/bin/X11 ~/bin . )
```

prompt Holds the primary prompt, similar to the bash **PS1** variable (page 319). If it is not set, the prompt is **>**, or **#** when you are working with **root** privileges. The shell expands an exclamation point in the prompt string to the current event number. The following is a typical line from a **.tcshrc** file that sets the value of **prompt**:

```
set prompt = '! $ '
```

Table 9-4 lists some of the formatting sequences you can use in **prompt** to achieve special effects.

Table 9-4 **prompt** formatting sequences

Sequence	Displays in prompt
%/	Value of **cwd** (the working directory)
%~	Same as **%/**, but replaces the path of the user's home directory with a tilde
%! *or* **%h** *or* **!**	Current event number
%d	Day of the week

Table 9-4 prompt formatting sequences (continued)

Sequence	Displays in prompt
%D	Day of the month
%m	Hostname without the domain
%M	Full hostname, including the domain
%n	User's username
%t	Time of day through the current minute
%p	Time of day through the current second
%W	Month as **mm**
%y	Year as **yy**
%Y	Year as **yyyy**
%#	A pound sign (**#**) if the user is running with **root** privileges; otherwise, a greater than sign (**>**)
%?	Exit status of the preceding command

prompt2 Holds the secondary prompt, which tcsh uses to indicate it is waiting for additional input. The default value is **%R?**. The TC Shell replaces **%R** with nothing when it is waiting for you to continue an unfinished command, the word **foreach** while iterating through a **foreach** structure (page 414), and the word **while** while iterating through a **while** structure (page 416).

When you press RETURN in the middle of a quoted string on a command line without ending the line with a backslash, tcsh displays an error message regardless of whether you use single or double quotation marks:

```
% echo "Please enter the three values
Unmatched ".
```

In the next example, the first RETURN is quoted (escaped); the shell interprets it literally. Under tcsh, single and double quotation marks produce the same result. The secondary prompt is a question mark (**?**).

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

prompt3 Holds the prompt used during automatic spelling correction. The default value is **CORRECT>%R (y|n|e|a)?**, where **R** is replaced by the corrected string.

savehist Specifies the number of commands saved from the history list when you log out. These events are saved in a file named **~/.history**. The shell uses these events as the initial history list when you log in again, causing your history list to persist across login sessions (page 385).

shell Holds the pathname of the shell you are using.

shlvl Holds the level of the shell. The TC Shell increments this variable each time you start a subshell and decrements it each time you exit a subshell. The TC Shell sets the value to 1 for a login shell.

status Holds the exit status returned by the last command. Similar to **$?** in bash (page 477).

tcsh Holds the version number of tcsh you are running.

time Provides two functions: automatic timing of commands using the time builtin and the format used by time. You can set this variable to either a single numeric value or an array holding a numeric value and a string. The numeric value is used to control automatic timing; any command that takes more than that number of CPU seconds to run has time display the command statistics when it finishes execution. When set to a value of 0 this results in statistics being displayed after every command. The string controls the formatting of the statistics using formatting sequences, including those listed in Table 9-5.

Table 9-5 time formatting sequences

Sequence	Displays
%U	Time the command spent running user code, in CPU seconds (user mode)
%S	Time the command spent running system code, in CPU seconds (kernel mode)
%E	Wall clock time (total elapsed) taken by the command
%P	Percentage of time the CPU spent on this task during this period, computed as (%U+%S)/%E
%W	Number of times the command's processes were swapped out to disk
%X	Average amount of shared code memory used by the command, in kilobytes
%D	Average amount of data memory used by the command, in kilobytes
%K	Total memory used by the command (as %X+%D), in kilobytes
%M	Maximum amount of memory used by the command, in kilobytes

Table 9-5 time formatting sequences (continued)

Sequence	Displays
%F	Number of major page faults (pages of memory that had to be read from disk)
%I	Number of input operations
%O	Number of output operations

By default the time builtin uses the string

```
"%Uu %Ss %E %P%  %X+%Dk %I+%Oio %Fpf+%Ww"
```

which generates output in the following format:

```
tcsh $ time
0.200u 0.340s 17:32:33.27 0.0%      0+0k 0+0io 1165pf+0w
```

You might want to time commands to check system performance. If commands consistently show many page faults and swaps, the system probably does not have enough memory; you should consider adding more. You can use the information that time reports to compare the performance of various system configurations and program algorithms.

tperiod Controls how often, in minutes, the shell executes the special **periodic** alias (page 387).

user The shell sets this variable to your username.

version The shell sets this variable to contain detailed information about the version of tcsh the system is running.

watch Set to an array of user and terminal pairs to watch for logins and logouts. The word **any** means any user or any terminal, so (**any any**) monitors all logins and logouts on all terminals, whereas (**zach ttyS1 any console $user any**) watches for **zach** on **ttyS1**, any user who accesses the system console, and any logins and logouts that use your account (presumably to catch intruders). By default logins and logouts are checked once every 10 minutes, but you can change this value by beginning the array with a numeric value giving the number of minutes between checks. If you set **watch** to (**1 any console**), logins and logouts by any user on the console will be checked once per minute. Reports are displayed just before a new shell prompt is issued. Also, the log builtin forces an immediate check whenever it is executed. See **who** (next) for information about how you can control the format of the **watch** messages.

who Controls the format of the information displayed in **watch** messages (Table 9-6).

Table 9-6 **who** formatting sequence

Sequence	Displays
%n	Username
%a	Action taken by the user
%l	Terminal on which the action took place

Table 9-6 **who** formatting sequence (continued)

Sequence	Displays
%M	Full hostname of remote host (or **local** if none) from which the action took place
$m	Hostname without domain name

The default string used for watch messages when **who** is unset is **"%n has %a %l from %m"**, which generates the following line:

```
sam has logged on tty2 from local
```

$** As in bash, this variable contains the PID number of the current shell; use it as **$$.

tcsh Variables That Act as Switches

The following shell variables act as switches; their values are not significant. If the variable has been declared, the shell takes the specified action. If not, the action is not taken or is negated. You can set these variables in a startup file, in a shell script, or from the command line.

autocorrect Causes the shell to attempt spelling correction automatically, just before each attempt at completion (page 394).

dunique Normally, pushd blindly pushes the new working directory onto the directory stack, meaning that you can end up with many duplicate entries on this stack. Set **dunique** to cause the shell to look for and delete any entries that duplicate the one it is about to push.

echo Causes the shell to display each command before it executes that command. Set **echo** by calling tcsh with the **–x** option or by using set.

filec Enables filename completion (page 391) when running tcsh as csh (and csh is linked to tcsh).

histlit Displays the commands in the history list exactly as entered, without interpretation by the shell (page 386).

ignoreeof Prevents you from using CONTROL-D to exit from a shell so you cannot accidentally log out. When this variable is declared, you must use **exit** or **logout** to leave a shell.

listjobs Causes the shell to list all jobs whenever a job is suspended.

listlinks Causes the ls–F builtin to show the type of file each symbolic link points to instead of marking the symbolic link with an @ symbol.

loginsh Set by the shell if the current shell is running as a login shell.

nobeep Disables all beeping by the shell.

noclobber Prevents you from accidentally overwriting a file when you redirect output and prevents you from creating a file when you attempt to append output to a nonexistent file (Table 9-7). To override **noclobber**, add an exclamation point to the symbol you use for redirecting or appending output (e.g., **>!** and **>>!**). For more information see page 143.

Table 9-7 How **noclobber** works

Command line	noclobber not declared	noclobber declared
x > ***fileout***	Redirects standard output from process **x** to ***fileout***. Overwrites ***fileout*** if it exists.	Redirects standard output from process **x** to ***fileout***. The shell displays an error message if ***fileout*** exists, and does not overwrite the file.
x >> ***fileout***	Redirects standard output from process **x** to ***fileout***. Appends new output to the end of ***fileout*** if it exists. Creates ***fileout*** if it does not exist.	Redirects standard output from process **x** to ***fileout***. Appends new output to the end of ***fileout*** if it exists. The shell displays an error message if ***fileout*** does not exist, and does not create the file.

noglob Prevents the shell from expanding ambiguous filenames. Allows you to use *, ?, ~, and [] literally on the command line or in a shell script without quoting them.

nonomatch Causes the shell to pass an ambiguous file reference that does not match a filename to the command being called. The shell does not expand the file reference. When you do not set **nonomatch**, tcsh generates a **No match** error message and does not execute the command.

```
tcsh $ cat questions?
cat: No match
tcsh $ set nonomatch
tcsh $ cat questions?
cat: questions?: No such file or directory
```

notify When set, tcsh sends a message to the screen immediately whenever a background job completes. Ordinarily tcsh notifies you about job completion just before displaying the next prompt. Refer to "Job Control" on page 304.

pushdtohome Causes a call to pushd without any arguments to change directories to your home directory (equivalent to **pushd –**).

pushdsilent Causes pushd and popd not to display the directory stack.

rmstar Causes the shell to request confirmation when you give an **rm *** command.

verbose Causes the shell to display each command after a history expansion (page 384). Set **verbose** by calling tcsh with the **–v** option or by using set.

visiblebell Causes audible beeps to be replaced by flashing the screen.

CONTROL STRUCTURES

The TC Shell uses many of the same control structures as the Bourne Again Shell. In each case the syntax is different, but the effects are the same. This section summarizes the differences between the control structures in the two shells. For more information refer to “Control Structures” on page 430.

if

The syntax of the **if** control structure is

if (expression) simple-command

The **if** control structure works only with simple commands, not with pipelines (page 145) or lists (page 149). You can use the **if...then** control structure (page 413) to execute more complex commands.

```
tcsh $ cat if_1
#!/bin/tcsh
# Routine to show the use of a simple if control structure.
#
if ( $#argv == 0 ) echo "if_1: There are no arguments."
```

The **if_1** script checks whether it was called with zero arguments. If the expression enclosed in parentheses evaluates to *true*—that is, if zero arguments were on the command line—the **if** structure displays a message.

In addition to logical expressions such as the one the **if_1** script uses, you can use expressions that return a value based on the status of a file. The syntax for this type of expression is

–n filename

where ***n*** is one of the values listed in Table 9-8.

If the result of the test is *true,* the expression has a value of 1; if it is *false,* the expression has a value of 0. If the specified file does not exist or is not accessible, tcsh evaluates the expression as 0. The following example checks whether the file specified on the command line is an ordinary or directory file (and not a device or other special file):

```
tcsh $ cat if_2
#!/bin/tcsh
if -f $1 echo "$1 is an ordinary or directory file."
```

Table 9-8 Value of *n*

n	Meaning
b	File is a block special file
c	File is a character special file
d	File is a directory file
e	File exists
f	File is an ordinary or directory file

Table 9-8 Value of *n* (continued)

n	Meaning
g	File has the set-group-ID bit set
k	File has the sticky bit (page 1126) set
l	File is a symbolic link
o	File is owned by user
p	File is a named pipe (FIFO)
r	The user has read access to the file
S	File is a socket special file
s	File is not empty (has nonzero size)
t	File descriptor (a single digit replacing ***filename***) is open and connected to the terminal
u	File has the set-user-ID bit set
w	User has write access to the file
X	File is either a builtin or an executable found by searching the directories in **$path**
x	User has execute access to the file
z	File is 0 bytes long

You can combine operators where it makes sense. For example, **–ox filename** is *true* if you own and have execute permission for the file. This expression is equivalent to **–o filename && –x filename**.

Some operators return useful information about a file other than reporting *true* or *false*. They use the same ***–n filename*** syntax, where ***n*** is one of the values shown in Table 9-9.

Table 9-9 Value of *n*

n	Meaning
A	The last time the file was accessed.*
A:	The last time the file was accessed displayed in a human-readable format.
M	The last time the file was modified.*
M:	The last time the file was modified displayed in a human-readable format.
C	The last time the file's inode was modified.*

Table 9-9 Value of *n* (continued)

n	Meaning
C:	The last time the file's inode was modified displayed in a human-readable format.
D	Device number for the file. This number uniquely identifies the device (a disk partition, for example) on which the file resides.
I	Inode number for the file. The inode number uniquely identifies a file on a particular device.
F	A string of the form **device:inode**. This string uniquely identifies a file anywhere on the system.
N	Number of hard links to the file.
P	The file's permissions, shown in octal, without a leading 0.
U	Numeric user ID of the file's owner.
U:	Username of the file's owner.
G	Numeric ID of the group the file is associated with.
G:	Name of the group the file is associated with.
Z	Number of bytes in the file.

*Time measured in seconds from the *epoch* (usually, the start of January 1, 1970).

You can use only one of these operators in a given test, and it must appear as the last operator in a multiple-operator sequence. Because 0 (zero) can be a valid response from some of these operators (for instance, the number of bytes in a file might be 0), most return –1 on failure instead of the 0 that the logical operators return on failure. The one exception is **F**, which returns a colon if it cannot determine the device and inode for the file.

When you want to use one of these operators outside of a control structure expression, you can use the filetest builtin to evaluate a file test and report the result:

```
tcsh $ filetest -z if_1
0
tcsh $ filetest -F if_1
2051:12694
tcsh $ filetest -Z if_1
131
```

goto

The **goto** statement has the following syntax:

*goto **label***

A **goto** builtin transfers control to the statement beginning with ***label:***. The following script fragment demonstrates the use of **goto**:

```
tcsh $ cat goto_1
#!/bin/tcsh
#
# test for 2 arguments
#
if ($#argv == 2) goto goodargs
echo "Usage: $0 arg1 arg2"
exit 1
goodargs:
...
```

The **goto_1** script displays a usage message (page 434) when it is called with more or fewer than two arguments.

INTERRUPT HANDLING

The **onintr** (on interrupt) statement transfers control when you interrupt a shell script. The syntax of an **onintr** statement is

*onintr **label***

When you press the interrupt key during execution of a shell script, the shell transfers control to the statement beginning with ***label:***. This statement allows you to terminate a script gracefully when it is interrupted. For example, you can use it to ensure that when you interrupt a shell script, the script removes temporary files before returning control to the parent shell.

The following script demonstrates the use of **onintr**. It loops continuously until you press the interrupt key, at which time it displays a message and returns control to the shell:

```
tcsh $ cat onintr_1
#!/bin/tcsh
# demonstration of onintr
onintr close
while ( 1 )
    echo "Program is running."
    sleep 2
end
close:
echo "End of program."
```

If a script creates temporary files, you can use **onintr** to remove them.

```
close:
rm -f /tmp/$$*
```

The ambiguous file reference **/tmp/$$*** matches all files in **/tmp** that begin with the PID number of the current shell. Refer to page 476 for a description of this technique for naming temporary files.

if...then...else

The **if...then...else** control structure has three forms. The first form, an extension of the simple **if** structure, executes more complex ***commands*** or a series of ***commands*** if ***expression*** is *true*. This form is still a one-way branch.

*if (**expression**) then*
  commands
endif

The second form is a two-way branch. If ***expression*** is *true*, the first set of ***commands*** is executed. If it is *false*, the set of ***commands*** following **else** is executed.

*if (**expression**) then*
  commands
else
  commands
endif

The third form is similar to the **if...then...elif** structure (page 436). It performs tests until it finds an ***expression*** that is *true* and then executes the corresponding ***commands***.

*if (**expression**) then*
  commands
*else if (**expression**) then*
  commands
. . .
else
  commands
endif

The following program assigns a value of 0, 1, 2, or 3 to the variable **class** based on the value of the first command-line argument. The program declares the variable **class** at the beginning for clarity; you do not need to declare it before its first use. Also for clarity, the script assigns the value of the first command-line argument to **number**.

```
tcsh $ cat if_else_1
#!/bin/tcsh
# routine to categorize the first
# command-line argument
set class
set number = $argv[1]
#
```

```
if ($number < 0) then
    @ class = 0
else if (0 <= $number && $number < 100) then
    @ class = 1
else if (100 <= $number && $number < 200) then
    @ class = 2
else
    @ class = 3
endif
#
echo "The number $number is in class $class."
```

The first **if** statement tests whether **number** is less than 0. If it is, the script assigns 0 to **class** and transfers control to the statement following **endif**. If it is not, the second **if** tests whether the number is between 0 and 100. The && Boolean AND operator yields a value of *true* if the expression on each side is *true*. If the number is between 0 and 100, 1 is assigned to **class** and control is transferred to the statement following **endif**. A similar test determines whether the number is between 100 and 200. If it is not, the final **else** assigns 3 to **class**. The **endif** closes the **if** control structure.

foreach

The **foreach** structure parallels the bash **for...in** structure (page 443). The syntax is

foreach ***loop-index (argument-list)***
  commands
end

This structure loops through ***commands***. The first time through the loop, the structure assigns the value of the first argument in ***argument-list*** to ***loop-index***. When control reaches the **end** statement, the shell assigns the value of the next argument from ***argument-list*** to ***loop-index*** and executes the commands again. The shell repeats this procedure until it exhausts ***argument-list***.

The following tcsh script uses a **foreach** structure to loop through the files in the working directory containing a specified string of characters in their filename and to change the string. For example, you can use it to change the string **memo** in filenames to **letter**. Thus, the filenames **memo.1**, **dailymemo**, and **memories** would be changed to **letter.1**, **dailyletter**, and **letterries**, respectively.

This script requires two arguments: the string to be changed (the old string) and the new string. The ***argument-list*** of the **foreach** structure uses an ambiguous file reference to loop through all files in the working directory with filenames that contain the first argument. For each filename that matches the ambiguous file reference, the mv utility changes the filename. The echo and sed commands appear within back ticks (`) that indicate command substitution: Executing the commands within the back ticks replaces the back ticks and everything between them. Refer to "Command Substitution" on page 371 for more information. The sed utility (page 669) substitutes the first argument with the second argument in the filename. The **$1** and **$2** are abbreviated forms of **$argv**[**1**] and **$argv**[**2**], respectively.

```
tcsh $ cat ren
#!/bin/tcsh
# Usage:        ren arg1 arg2
#               changes the string arg1 in the names of files
#               in the working directory into the string arg2
if ($#argv != 2) goto usage
foreach i ( *$1* )
    mv $i `echo $i | sed -n s/$1/$2/p`
end
exit 0

usage:
echo "Usage: ren arg1 arg2"
exit 1
```

optional The next script uses a **foreach** loop to assign the command-line arguments to the elements of an array named **buffer**:

```
tcsh $ cat foreach_1
#!/bin/tcsh
# routine to zero-fill argv to 20 arguments
#
set buffer = (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
set count = 1
#
if ($#argv > 20) goto toomany
#
foreach argument ($argv[*])
    set buffer[$count] = $argument
    @ count++
end
# REPLACE command ON THE NEXT LINE WITH
# THE PROGRAM YOU WANT TO CALL.
exec command $buffer[*]
#
toomany:
echo "Too many arguments given."
echo "Usage: foreach_1 [up to 20 arguments]"
exit 1
```

The **foreach_1** script calls another program named **command** with a command line guaranteed to contain 20 arguments. If **foreach_1** is called with fewer than 20 arguments, it fills the command line with zeros to complete the 20 arguments for **command**. Providing more than 20 arguments causes it to display a usage message and exit with an error status of 1.

The **foreach** structure loops through the commands one time for each command-line argument. Each time through the loop, **foreach** assigns the value of the next argument from the command line to the variable **argument**. Then the script assigns each of these values to an element of the array **buffer**. The variable **count** maintains the index for the **buffer** array. A postfix operator increments the **count** variable using @ (**@ count++**). The exec builtin (bash and tcsh; page 493) calls **command** so a new process is not initiated. (Once **command** is called, the process running this routine is no longer needed so a new process is not required.)

while

The syntax of the **while** structure is

*while (**expression**)*
    commands
end

This structure continues to loop through ***commands*** while ***expression*** is *true*. If ***expression*** is *false* the first time it is evaluated, the structure never executes ***commands***.

```
tcsh $ cat while_1
#!/bin/tcsh
# Demonstration of a while control structure.
# This routine sums the numbers between 1 and n;
# n is the first argument on the command line.
#
set limit = $argv[1]
set index = 1
set sum = 0
#
while ($index <= $limit)
    @ sum += $index
    @ index++
end
#
echo "The sum is $sum"
```

This program computes the sum of all integers up to and including ***n***, where ***n*** is the first argument on the command line. The **+=** operator assigns the value of **sum + index** to **sum**.

break AND continue

You can interrupt a **foreach** or **while** structure with a **break** or **continue** statement. These statements execute the remaining commands on the line before they transfer control. The **break** statement transfers control to the statement after the **end** statement, terminating execution of the loop. The **continue** statement transfers control to the **end** statement, which continues execution of the loop.

switch

The **switch** structure is analogous to the bash **case** structure (page 454):

> *switch (**test-string**)*
>
> > *case **pattern:***
> > > ***commands***
> >
> > *breaksw*
> >
> > *case **pattern:***
> > > ***commands***
> >
> > *breaksw*
> > ...
> > *default:*
> > > ***commands***
> >
> > *breaksw*
>
> *endsw*

The **breaksw** statement transfers control to the statement following the **endsw** statement. If you omit **breaksw**, control falls through to the next command. You can use any of the special characters listed in Table 10-2 on page 456 within ***pattern*** except the pipe symbol (|).

```
tcsh $ cat switch_1
#!/bin/tcsh
# Demonstration of a switch control structure.
# This routine tests the first command-line argument
# for yes or no in any combination of uppercase and
# lowercase letters.
#
# test that argv[1] exists
if ($#argv != 1) then
    echo "Usage: $0 [yes|no]"
    exit 1
else
    # argv[1] exists, set up switch based on its value
    switch ($argv[1])
    # case of YES
        case [yY][eE][sS]:
        echo "Argument one is yes."
        breaksw
    #
    # case of NO
        case [nN][oO]:
        echo "Argument one is no."
    breaksw
    #
    # default case
```

```
            default:
            echo "Argument one is not yes or no."
            breaksw
        endsw
    endif
```

Builtins

Builtins are commands that are part of (built into) the shell. When you give a simple filename as a command, the shell first checks whether it is the name of a builtin. If it is, the shell executes it as part of the calling process; the shell does not fork a new process to execute the builtin. The shell does not need to search the directory structure for builtin programs because they are immediately available to the shell.

If the simple filename you give as a command is not a builtin, the shell searches the directory structure for the program you want, using the **PATH** variable as a guide. When it finds the program, the shell forks a new process to execute the program.

Although they are not listed in Table 9-10, the control structure keywords (**if**, **foreach**, **endsw**, and so on) are builtins. Table 9-10 describes many of the tcsh builtins, some of which are also built into other shells.

Table 9-10 tcsh builtins

Builtin	Function
% ***job***	A synonym for the fg builtin. The ***job*** is the job number of the job you want to bring to the foreground (page 305).
% ***job*** &	A synonym for the bg builtin. The ***job*** is the number of the job you want to put in the background (page 306).
@	Similar to the set builtin but evaluates numeric expressions. Refer to "Numeric Variables" on page 398.
alias	Creates and displays aliases; tcsh uses a different syntax than bash. Refer to "Aliases" on page 387.
alloc	Displays a report of the amount of free and used memory.
bg	Moves a suspended job into the background (page 306).
bindkey	Controls the mapping of keys to the tcsh command-line editor commands.
bindkey	Without any arguments, bindkey lists all key bindings (page 393).
bindkey –l	Lists all available editor commands and gives a short description of each.
bindkey –e	Puts the command-line editor in emacs mode (page 393).
bindkey –v	Puts the command-line editor in vi(m) mode (page 393).
bindkey ***key command***	Attaches the editor command ***command*** to the key ***key***.

Table 9-10 tcsh builtins (continued)

Builtin	Function
bindkey –b ***key command***	Similar to the previous form but allows you to specify CONTROL keys by using the form C–x (where x is the character you type while you press the CONTROL key), specify META key sequences as M-x (on most keyboards used with Linux, the ALT key is the META key; macOS uses the OPTION key [but you have to set an option in the Terminal utility to use it; see page 1076]), and specify function keys as F-x.
bindkey –c ***key command***	Binds the key ***key*** to the command ***command***. Here the ***command*** is not an editor command but rather a shell builtin or an executable program.
bindkey –s ***key string***	Causes tcsh to substitute ***string*** when you press ***key***.
builtins	Displays a list of all builtins.
cd or chdir	Changes the working directory (page 94).
dirs	Displays the directory stack (page 307).
echo	Displays its arguments. You can prevent echo from displaying a RETURN at the end of a line by using the **–n** option (see "Reading User Input" on page 401) or by using a trailing **\c** (see "read: Accepts User Input" on page 489). The echo builtin is similar to the echo utility (page 812).
eval	Scans and evaluates the command line. When you put eval in front of a command, the command is scanned twice by the shell before it is executed. This feature is useful with a command that is generated through command or variable substitution. Because of the order in which the shell processes a command line, it is sometimes necessary to repeat the scan to achieve the desired result (page 500).
exec	Overlays the program currently being executed with another program in the same shell. The original program is lost. Refer to "exec: Executes a Command or Redirects File Descriptors" on page 493 for more information; also refer to source on page 421.
exit	Exits from a TC Shell. When you follow exit with a numeric argument, tcsh returns that number as the exit status (page 477).
fg	Moves a job into the foreground (page 304).
filetest	Takes one of the file inquiry operators followed by one or more filenames and applies the operator to each filename (page 411). Returns the results as a SPACE-separated list.

Table 9-10 tcsh builtins (continued)

Builtin	Function
glob	Like echo, but does not display SPACEs between its arguments and does not follow its display with a NEWLINE.
hashstat	Reports on the efficiency of tcsh's hash mechanism. The hash mechanism speeds the process of searching through the directories in your search path. See also rehash (page 421) and unhash (page 422).
history	Displays a list of recent commands (page 384).
jobs	Displays a list of jobs (suspended commands and those running in the background).
kill	Terminates a job or process (page 499).
limit	Limits the computer resources that the current process and any processes it creates can use. You can put limits on the number of seconds of CPU time the process can use, the size of files that the process can create, and so forth.
log	Immediately produces the report that the watch shell variable (page 406) would normally cause tcsh to produce every 10 minutes.
login	Logs in a user. Can be followed by a username.
logout	Ends a session if you are using a login shell.
ls–F	Similar to **ls –F** (page 885) but faster. (This builtin is the characters **ls–F** without any SPACEs.)
nice	Lowers the processing priority of a command or a shell. This builtin is useful if you want to run a command that makes large demands on the system and you do not need the output right away. If you are working with **root** privileges, you can use nice to raise the priority of a command. Refer to page 916 for more information on the nice builtin and the nice utility.
nohup	Allows you to log out without terminating processes running in the background. Some systems are set up this way by default. Refer to page 920 for information on the nohup builtin and the nohup utility.
notify	Causes the shell to notify you immediately when the status of one of your jobs changes (page 304).
onintr	Controls the action an interrupt causes within a script (page 412). See "trap: Catches a Signal" on page 496 for information on the equivalent command in bash.
popd	Changes the working directory to the directory on the top of the directory stack and removes that directory from the directory stack (page 307).
printenv	Displays all environment variable names and values.

Table 9-10 tcsh builtins (continued)

Builtin	Function
pushd	Changes the working directory and places the new directory at the top of the directory stack (page 308).
rehash	Re-creates the internal tables used by the hash mechanism. Whenever a new instance of tcsh is invoked, the hash mechanism creates a sorted list of all available commands based on the value of **path**. After you add a command to a directory in **path**, use rehash to re-create the sorted list of commands. If you do not, tcsh might not be able to find the new command. Also refer to hashstat (page 420) and unhash (page 422).
repeat	Takes two arguments—a count and a simple command (no pipelines or lists)—and repeats the command the number of times specified by the count.
sched	Executes a command at a specified time. For example, the following command causes the shell to display the message **Dental appointment.** at 10 AM: `tcsh $ sched 10:00 echo "Dental appointment."` Without any arguments, sched displays the list of scheduled commands. When the time to execute a scheduled command arrives, tcsh executes the command just before it displays a prompt.
set	Declares, initializes, and displays local variables (page 396).
setenv	Declares, initializes, and displays environment variables (page 396).
shift	Analogous to the bash shift builtin (page 473). Without an argument, shift promotes the indexes of the **argv** array. You can use it with an argument of an array name to perform the same operation on that array.
source	Executes the shell script given as its argument: source does not fork another process. It is similar to the bash . (dot) builtin (page 290). The source builtin expects a TC Shell script so no leading **#!** is required in the script. The current shell executes source; thus the script can contain commands, such as set, that affect the current shell. After you make changes to your **.tcshrc** or **.login** file, you can use source to execute it from the shell, thereby putting the changes into effect without logging out and in. You can nest source builtins.
stop	Stops a job or process that is running in the background. The stop builtin accepts multiple arguments.
suspend	Stops the current shell and puts it in the background. It is similar to the suspend key, which stops jobs running in the foreground. This builtin will not suspend a login shell.
time	Executes the command you give it as an argument. It displays a summary of time-related information about the executed command, according to the **time** shell variable (page 405). Without an argument, time displays the times for the current shell and its children.

Table 9-10 tcsh builtins (continued)

Builtin	Function
umask	Identifies or changes the access permissions that tcsh assigns to files you create (page 1021).
unalias	Removes an alias (page 387).
unhash	Turns off the hash mechanism. See also hashstat (page 420) and rehash (page 421).
unlimit	Removes limits (page 420) on the current process.
unset	Removes a variable declaration (page 396).
unsetenv	Removes an environment variable declaration (page 396).
wait	Causes the shell to wait for all child processes to terminate. When you give a wait command in response to a shell prompt, tcsh does not display a prompt until all background processes have finished execution. If you interrupt it with the interrupt key, wait displays a list of background processes before tcsh displays a prompt.
where	When given the name of a command as an argument, locates all occurrences of the command and, for each, tells you whether it is an alias, a builtin, or an executable program in your path.
which	Similar to where but reports on only the command that would be executed, not all occurrences. This builtin is much faster than the which utility and reports on aliases and builtins.

Chapter Summary

Like the Bourne Again Shell, the TC Shell is both a command interpreter and a programming language. The TC Shell, which is based on the C Shell that was developed at the University of California at Berkeley, includes popular features such as history, alias, and job control.

You might prefer to use tcsh as a command interpreter, especially if you are familiar with the C Shell. You can use chsh to change your login shell to tcsh. However, running tcsh as your interactive shell does *not* cause tcsh to run shell scripts; they will continue to be run by bash unless you explicitly specify another shell on the first line of the script or specify the script name as an argument to tcsh. Specifying the shell on the first line of a shell script ensures the behavior you expect.

If you are familiar with bash, you will notice some differences between the two shells. For instance, the syntax you use to assign a value to a variable differs. In addition, tcsh allows SPACEs around the equal sign. Both numeric and nonnumeric variables are created and given values using the set builtin. The @ builtin can evaluate numeric expressions for assignment to numeric variables.

setenv Because there is no export builtin in tcsh, you must use the setenv builtin to create an environment (global) variable. You can also assign a value to the variable with the setenv command. The command unset removes both shell and environment variables, whereas the command unsetenv removes only environment variables.

Aliases The syntax of the tcsh alias builtin is slightly different from that of alias in bash. Unlike bash, the tcsh aliases permit you to substitute command-line arguments using the history mechanism syntax.

Most other tcsh features, such as history, word completion, and command-line editing, closely resemble their bash counterparts. The syntax of the tcsh control structures is slightly different but provides functionality equivalent to that found in bash.

Globbing The term *globbing*, a carryover from the original Bourne Shell, refers to the matching of strings containing special characters (such as * and ?) to filenames. If tcsh is unable to generate a list of filenames matching a globbing pattern, it displays an error message. This behavior contrasts with that of bash, which simply leaves the pattern alone.

Standard input and standard output can be redirected in tcsh, but there is no straightforward way to redirect them independently. Doing so requires the creation of a subshell that redirects standard output to a file while making standard error available to the parent process.

EXERCISES

1. Assume that you are working with the following history list:

```
37  mail zach
38  cd /home/sam/correspondence/business/cheese_co
39  less letter.0321
40  vim letter.0321
41  cp letter.0321 letter.0325
42  grep hansen letter.0325
43  vim letter.0325
44  lpr letter*
45  cd ../milk_co
46  pwd
47  vim wilson.0321 wilson.0329
```

   Using the history mechanism, give commands to

   a. Send mail to Zach.

b. Use vim to edit a file named **wilson.0329**.

c. Send **wilson.0329** to the printer.

d. Send both **wilson.0321** and **wilson.0329** to the printer.

2. a. How can you display the aliases currently in effect?

   b. Write an alias named **homedots** that lists the names (only) of all hidden files in your home directory.

3. a. How can you prevent a command from sending output to the terminal when you start it in the background?

   b. What can you do if you start a command in the foreground and later decide that you want it to run in the background?

4. Which statement can you put in your **~/.tcshrc** file to prevent accidentally overwriting a file when you redirect output? How can you override this feature?

5. Assume that the working directory contains the following files:

```
adams.ltr.03
adams.brief
adams.ltr.07
abelson.09
abelson.brief
anthony.073
anthony.brief
azevedo.99
```

   What happens if you press TAB after typing the following commands?

   a. **less adams.l**

   b. **cat a**

   c. **ls ant**

   d. **file az**

   What happens if you press CONTROL-D after typing the following commands?

   e. **ls ab**

   f. **less a**

6. Write an alias named **backup** that takes a filename as an argument and creates a copy of that file with the same name and a filename extension of **.bak**.

7. Write an alias named **qmake** (quiet make) that runs make with both standard output and standard error redirected to the file named **make.log**. The command **qmake** should accept the same options and arguments as make.

8. How can you make tcsh always display the pathname of the working directory as part of its prompt?

Advanced Exercises

9. Which lines do you need to change in the Bourne Again Shell script **command_menu** (page 456) to turn it into a TC Shell script? Make the changes and verify that the new script works.

10. Users often find rm (and even **rm –i**) too unforgiving because this utility removes files irrevocably. Create an alias named **delete** that moves files specified by its argument(s) into the **~/.trash** directory. Create a second alias named **undelete** that moves a file from the **~/.trash** directory into the working directory. Put the following line in your **~/.logout** file to remove any files that you deleted during the login session:

    ```
    /bin/rm -f $HOME/.trash/* >& /dev/null
    ```

    Explain what could be different if the following line were put in your **~/.logout** file instead:

    ```
    rm $HOME/.trash/*
    ```

11. Modify the **foreach_1** script (page 415) so that it takes the command to exec as an argument.

12. Rewrite the program **while_1** (page 416) so that it runs faster. Use the time builtin to verify the improvement in execution time.

13. Write your own version of find named **myfind** that writes output to the file **findout** but without the clutter of error messages, such as those generated when you do not have permission to search a directory. The **myfind** command should accept the same options and arguments as find. Can you think of a situation in which **myfind** does not work as desired?

14. When the **foreach_1** script (page 415) is supplied with 20 or fewer arguments, why are the commands following **toomany:** not executed? (Why is there no exit command?)

This page intentionally left blank

PART IV

Programming Tools

This page intentionally left blank

10

Programming the Bourne Again Shell (bash)

In This Chapter

Objectives

After reading this chapter you should be able to:

- Use control structures to implement decision making and repetition in shell scripts
- Handle input to and output from scripts
- Use shell variables (local) and environment variables (global)
- Evaluate the value of numeric variables
- Use bash builtin commands to call other scripts inline, trap signals, and kill processes
- Use arithmetic and logical expressions
- List standard programming practices that result in well-written scripts

Chapter 5 introduced the shells and Chapter 8 went into detail about the Bourne Again Shell. This chapter introduces additional Bourne Again Shell commands, builtins, and concepts that carry shell programming to a point where it can be useful. Although you might make use of shell programming as a system administrator, you do not have to read this chapter to perform system administration tasks. Feel free to skip this chapter and come back to it if and when you like.

The first part of this chapter covers programming control structures, also called control flow constructs. These structures allow you to write scripts that can loop over command-line arguments, make decisions based on the value of a variable, set up menus, and more. The Bourne Again Shell uses the same constructs found in programming languages such as C.

The next part of this chapter discusses parameters and variables, going into detail about array variables, shell versus environment variables, special parameters, and positional parameters. The exploration of builtin commands covers type, which displays information about a command, and read, which allows a shell script to accept user input. The section on the exec builtin demonstrates how to use exec to execute a command efficiently by replacing a process and explains how to use exec to redirect input and output from within a script.

The next section covers the trap builtin, which provides a way to detect and respond to operating system signals (such as the signal generated when you press CONTROL-C). The discussion of builtins concludes with a discussion of kill, which can abort a process, and getopts, which makes it easy to parse options for a shell script. Table 10-6 on page 504 lists some of the more commonly used builtins.

Next, the chapter examines arithmetic and logical expressions as well as the operators that work with them. The final section walks through the design and implementation of two major shell scripts.

This chapter contains many examples of shell programs. Although they illustrate certain concepts, most use information from earlier examples as well. This overlap not only reinforces your overall knowledge of shell programming but also demonstrates how you can combine commands to solve complex tasks. Running, modifying, and experimenting with the examples in this book is a good way to become comfortable with the underlying concepts.

Do not name a shell script test

tip You can unwittingly create a problem if you name a shell script **test** because a bash builtin has the same name. Depending on how you call your script, you might run either your script or the builtin, leading to confusing results.

Control Structures

The *control flow* commands alter the order of execution of commands within a shell script. The TC Shell uses a different syntax for these commands (page 408) than the Bourne Again Shell does. Control structures include the **if...then**, **for...in**, **while**, **until**, and **case** statements. In addition, the **break** and **continue** statements work in conjunction with the control structures to alter the order of execution of commands within a script.

Getting help with control structures

You can use the bash **help** command to display information about bash control structures. See page 39 for more information.

if...then

The **if...then** control structure has the following syntax (see page 409 for tcsh):

if ***test-command***
    then
        commands
fi

The ***bold*** words in the syntax description are the items you supply to cause the structure to have the desired effect. The *nonbold* words are the keywords the shell uses to identify the control structure.

test builtin

Figure 10-1 shows that the **if** statement tests the status returned by the ***test-command*** and transfers control based on this status. The end of the **if** structure is marked by a **fi** statement (*if* spelled backward). The following script prompts for two words, reads them, and then uses an **if** structure to execute commands based on the result returned by the test builtin (tcsh uses the test utility) when it compares the two words. (See page 1005 for information on the test utility, which is similar to the test builtin.) The

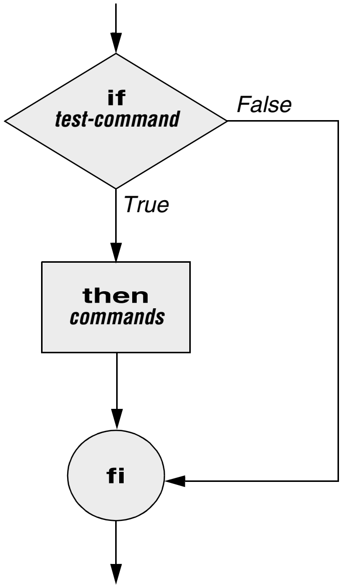

Figure 10-1 An **if...then** flowchart

test builtin returns a status of *true* if the two words are the same and *false* if they are not. Double quotation marks around **$word1** and **$word2** make sure test works properly if you enter a string that contains a SPACE or other special character.

```
$ cat if1
read -p "word 1: " word1
read -p "word 2: " word2

if test "$word1" = "$word2"
    then
        echo "Match"
fi
echo "End of program."

$ ./if1
word 1: peach
word 2: peach
Match
End of program.
```

In the preceding example the ***test-command*** is **test "$word1" = "$word2"**. The test builtin returns a *true* status if its first and third arguments have the relationship specified by its second argument. If this command returns a *true* status (= 0), the shell executes the commands between the **then** and **fi** statements. If the command returns a *false* status (not = 0), the shell passes control to the statement following **fi** without executing the statements between **then** and **fi**. The effect of this **if** statement is to display **Match** if the two words are the same. The script always displays **End of program.**

Builtins In the Bourne Again Shell, test is a builtin—part of the shell. It is also a stand-alone utility kept in **/usr/bin/test**. This chapter discusses and demonstrates many Bourne Again Shell builtins. Each bash builtin might or might not be a builtin in tcsh. The shell will use the builtin version if it is available and the utility if it is not. Each version of a command might vary slightly from one shell to the next and from the utility to any of the shell builtins. See page 489 for more information on shell builtins.

Checking arguments The next program uses an **if** structure at the beginning of a script to confirm that you have supplied at least one argument on the command line. The test **–eq** criterion compares two integers; the shell expands the **$#** special parameter (page 475) to the number of command-line arguments. This structure displays a message and exits from the script with an exit status of 1 if you do not supply at least one argument.

```
$ cat chkargs
if test $# -eq 0
    then
        echo "You must supply at least one argument."
        exit 1
fi
echo "Program running."
```

```
$ ./chkargs
You must supply at least one argument.
$ ./chkargs abc
Program running.
```

A test like the one shown in **chkargs** is a key component of any script that requires arguments. To prevent the user from receiving meaningless or confusing information from the script, the script needs to check whether the user has supplied the appropriate arguments. Some scripts simply test whether arguments exist (as in **chkargs**); other scripts test for a specific number or specific kinds of arguments.

You can use test to verify the status of a file argument or the relationship between two file arguments. After verifying that at least one argument has been given on the command line, the following script tests whether the argument is the name of an ordinary file (not a directory or other type of file). The test builtin with the **–f** criterion and the first command-line argument (**$1**) checks the file.

```
$ cat is_ordfile
if test $# -eq 0
    then
        echo "You must supply at least one argument."
        exit 1
fi
if test -f "$1"
    then
        echo "$1 is an ordinary file."
    else
        echo "$1 is NOT an ordinary file."
fi
```

You can test many other characteristics of a file using test criteria; see Table 10-1.

Table 10-1 test builtin criteria

Criterion	Tests file to see if it
–d	Exists and is a directory file
–e	Exists
–f	Exists and is an ordinary file (not a directory)
–r	Exists and is readable
–s	Exists and has a size greater than 0 bytes
–w	Exists and is writable
–x	Exists and is executable

Other test criteria provide ways to test relationships between two files, such as whether one file is newer than another. Refer to examples later in this chapter and to test on page 1005 for more information.

Always test the arguments

tip To keep the examples in this book short and focused on specific concepts, the code to verify arguments is often omitted or abbreviated. It is good practice to test arguments in shell programs that other people will use. Doing so results in scripts that are easier to debug, run, and maintain.

[] is a synonym for test

The following example—another version of **chkargs**—checks for arguments in a way that is more traditional for Linux shell scripts. This example uses the bracket ([]) synonym for test. Rather than using the word test in scripts, you can surround the arguments to test with brackets. The brackets must be surrounded by whitespace (SPACEs or TABs).

```
$ cat chkargs2
if [ $# -eq 0 ]
    then
        echo "Usage: chkargs2 argument..." 1>&2
        exit 1
fi
echo "Program running."
exit 0

$ ./chkargs2
Usage: chkargs2 argument...
$ ./chkargs2 abc
Program running.
```

Usage messages

The error message that **chkargs2** displays is called a *usage message* and uses the **1>&2** notation to redirect its output to standard error (page 294). After issuing the usage message, **chkargs2** exits with an exit status of 1, indicating an error has occurred. The **exit 0** command at the end of the script causes **chkargs2** to exit with a 0 status after the program runs without an error. The Bourne Again Shell returns the exit status of the last command the script ran if you omit the status code.

The usage message is commonly used to specify the type and number of arguments the script requires. Many Linux utilities provide usage messages similar to the one in **chkargs2**. If you call a utility or other program with the wrong number or wrong kind of arguments, it will often display a usage message. Following is the usage message that cp displays when you call it with only one argument:

```
$ cp a
cp: missing destination file operand after 'a'
Try 'cp --help' for more information.
```

if...then...else

The introduction of an **else** statement turns the **if** structure into the two-way branch shown in Figure 10-2. The **if...then...else** control structure (available in tcsh with a slightly different syntax) has the following syntax:

if ***test-command***
    then
        commands
    else
        commands
fi

Because a semicolon (**;**) ends a command just as a NEWLINE does, you can place **then** on the same line as **if** by preceding it with a semicolon. (Because **if** and **then** are separate builtins, they require a control operator between them; a semicolon and NEWLINE work equally well [page 300].) Some people prefer this notation for aesthetic reasons; others like it because it saves space.

if ***test-command****; then*
        commands
    else
        commands
fi

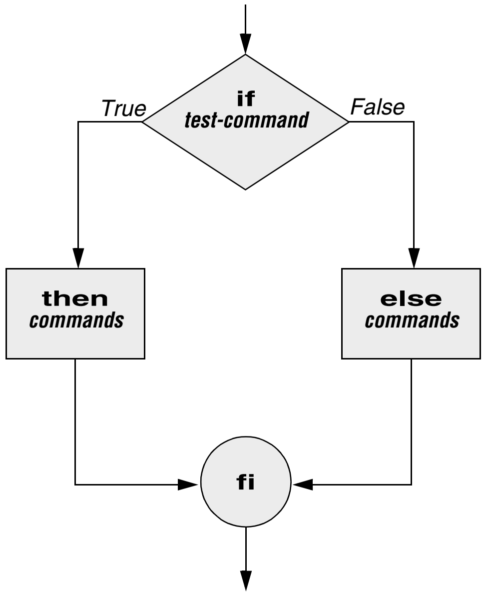

Figure 10-2 An **if...then...else** flowchart

If the ***test-command*** returns a *true* status, the **if** structure executes the commands between the **then** and **else** statements and then diverts control to the statement following **fi**. If the ***test-command*** returns a *false* status, the **if** structure executes the commands following the **else** statement.

When you run the **out** script with arguments that are filenames, it displays the files on the terminal. If the first argument is **–v** (called an option in this case), **out** uses less (page 53) to display the files one screen at a time. After determining that it was called with at least one argument, **out** tests its first argument to see whether it is **–v**. If the result of the test is *true* (the first argument is **–v**), **out** uses the shift builtin (page 473) to shift the arguments to get rid of the **–v** and displays the files using less. If the result of the test is *false* (the first argument is *not* **–v**), the script uses cat to display the files.

```
$ cat out
if [ $# -eq 0 ]
    then
        echo "Usage: $0 [-v] filenames..." 1>&2
        exit 1
fi

if [ "$1" = "-v" ]
    then
        shift
        less -- "$@"
    else
        cat -- "$@"
fi
```

optional In **out**, the **––** argument to cat and less tells these utilities that no more options follow on the command line and not to consider leading hyphens (**–**) in the following list as indicating options. Thus, **––** allows you to view a file whose name starts with a hyphen (page 133). Although not common, filenames beginning with a hyphen do occasionally occur. (You can create such a file by using the command **cat > –fname**.) The **––** argument works with all Linux utilities that use the getopts builtin (page 501) to parse their options; it does not work with more and a few other utilities. This argument is particularly useful when used in conjunction with rm to remove a file whose name starts with a hyphen (**rm –– –fname**), including any you create while experimenting with the **––** argument.

if...then...elif

The **if...then...elif** control structure (Figure 10-3; not in tcsh) has the following syntax:

if ***test-command***
    then
        commands

elif ***test-command***
then
commands
. . .
else
commands
fi

The **elif** statement combines the **else** statement and the **if** statement and enables you to construct a nested set of **if...then...else** structures (Figure 10-3). The difference between the **else** statement and the **elif** statement is that each **else** statement must be paired with a **fi** statement, whereas multiple nested **elif** statements require only a single closing **fi** statement.

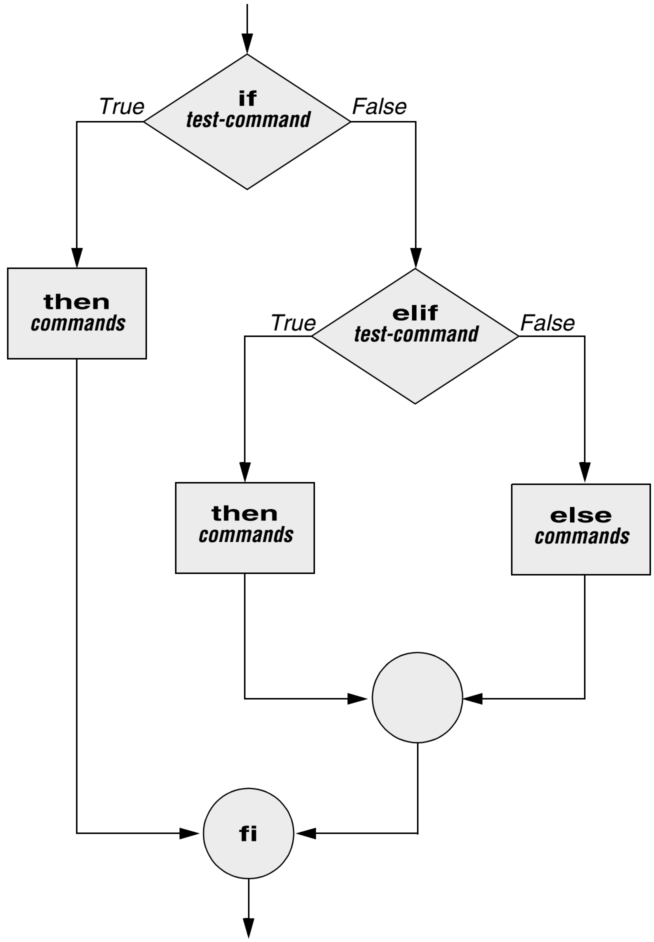

Figure 10-3 An **if...then...elif** flowchart

The following example shows an **if...then...elif** control structure. This shell script compares three words the user enters. The first **if** statement uses the Boolean AND operator (**–a**) as an argument to test. The test builtin returns a *true* status if the first and second logical comparisons are *true* (that is, **word1** matches **word2** and **word2** matches **word3**). If test returns a *true* status, the script executes the command following the next **then** statement, passes control to the statement following **fi**, and terminates.

```
$ cat if3
read -p "word 1: " word1
read -p "word 2: " word2
read -p "word 3: " word3
if [ "$word1" = "$word2" -a "$word2" = "$word3" ]
    then
        echo "Match: words 1, 2, & 3"
    elif [ "$word1" = "$word2" ]
    then
        echo "Match: words 1 & 2"
    elif [ "$word1" = "$word3" ]
    then
        echo "Match: words 1 & 3"
    elif [ "$word2" = "$word3" ]
    then
        echo "Match: words 2 & 3"
    else
        echo "No match"
fi


$ ./if3
word 1: apple
word 2: orange
word 3: pear
No match
$ ./if3
word 1: apple
word 2: orange
word 3: apple
Match: words 1 & 3
$ ./if3
word 1: apple
word 2: apple
word 3: apple
Match: words 1, 2, & 3
```

If the three words are not the same, the structure passes control to the first **elif**, which begins a series of tests to see if any pair of words is the same. As the nesting continues, if any one of the **elif** statements is satisfied, the structure passes control to the next **then** statement and subsequently to the statement following **fi**. Each time an **elif** statement is not satisfied, the structure passes control to the next **elif** statement. The double quotation marks around the arguments to echo that contain ampersands (&) prevent the shell from interpreting the ampersands as special characters.

optional THE lnks SCRIPT

The following script, named **lnks**, demonstrates the **if...then** and **if...then...elif** control structures. This script finds hard links to its first argument: a filename. If you provide the name of a directory as the second argument, **lnks** searches for links in the directory hierarchy rooted at that directory. If you do not specify a directory, **lnks** searches the working directory and its subdirectories. This script does not locate symbolic links.

```
$ cat lnks
#!/bin/bash
# Identify links to a file
# Usage: lnks file [directory]

if [ $# -eq 0 -o $# -gt 2 ]; then
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
fi
if [ -d "$1" ]; then
    echo "First argument cannot be a directory." 1>&2
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
else
    file="$1"
fi
if [ $# -eq 1 ]; then
        directory="."
    elif [ -d "$2" ]; then
        directory="$2"
    else
        echo "Optional second argument must be a directory." 1>&2
        echo "Usage: lnks file [directory]" 1>&2
        exit 1
fi

# Check that file exists and is an ordinary file
if [ ! -f "$file" ]; then
    echo "lnks: $file not found or is a special file" 1>&2
    exit 1
fi
# Check link count on file
set -- $(ls -l "$file")

linkcnt=$2
if [ "$linkcnt" -eq 1 ]; then
    echo "lnks: no other hard links to $file" 1>&2
    exit 0
fi

# Get the inode of the given file
set $(ls -i "$file")

inode=$1

# Find and print the files with that inode number
echo "lnks: using find to search for links..." 1>&2
find "$directory" -xdev -inum $inode -print
```

Max has a file named **letter** in his home directory. He wants to find links to this file in his and other users' home directory file hierarchies. In the following example, Max calls **lnks** from his home directory to perform the search. If you are running macOS, substitute **/Users** for **/home**. The second argument to **lnks**, **/home**, is the pathname of the directory where Max wants to start the search. The **lnks** script reports that **/home/max/letter** and **/home/zach/draft** are links to the same file:

```
$ ./lnks letter /home
lnks: using find to search for links...
/home/max/letter
/home/zach/draft
```

In addition to the **if...then...elif** control structure, **lnks** introduces other features that are commonly used in shell programs. The following discussion describes **lnks** section by section.

Specify the shell The first line of the **lnks** script uses **#!** (page 297) to specify the shell that will execute the script:

```
#!/bin/bash
```

In this chapter, the **#!** notation appears only in more complex examples. It ensures that the proper shell executes the script, even when the user is running a different shell or the script is called from a script running a different shell.

Comments The second and third lines of **lnks** are comments; the shell ignores text that follows a hashmark (**#**) up to the next NEWLINE character. These comments in **lnks** briefly identify what the file does and explain how to use it:

```
# Identify links to a file
# Usage: lnks file [directory]
```

Usage messages The first **if** statement tests whether **lnks** was called with zero arguments or more than two arguments:

```
if [ $# -eq 0 -o $# -gt 2 ]; then
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
fi
```

If either of these conditions is *true*, **lnks** sends a usage message to standard error and exits with a status of 1. The double quotation marks around the usage message prevent the shell from interpreting the brackets as special characters. The brackets in the usage message indicate that the **directory** argument is optional.

The second **if** statement tests whether the first command-line argument (**$1**) is a directory (the **–d** argument to test returns *true* if the file exists and is a directory):

```
if [ -d "$1" ]; then
    echo "First argument cannot be a directory." 1>&2
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
else
    file="$1"
fi
```

If the first argument is a directory, **lnks** displays a usage message and exits. If it is not a directory, **lnks** saves the value of **$1** in the **file** variable because later in the script set resets the command-line arguments. If the value of **$1** is not saved before the set command is issued, its value is lost.

Test the arguments The next section of **lnks** is an **if...then...elif** statement:

```
if [ $# -eq 1 ]; then
        directory="."
    elif [ -d "$2" ]; then
        directory="$2"
    else
        echo "Optional second argument must be a directory." 1>&2
        echo "Usage: lnks file [directory]" 1>&2
        exit 1
fi
```

The first ***test-command*** determines whether the user specified a single argument on the command line. If the ***test-command*** returns 0 (*true*), the **directory** variable is assigned the value of the working directory (**.**). If the ***test-command*** returns a nonzero value (*false*), the **elif** statement tests whether the second argument is a directory. If it is a directory, the **directory** variable is set equal to the second command-line argument, **$2**. If **$2** is not a directory, **lnks** sends a usage message to standard error and exits with a status of 1.

The next **if** statement in **lnks** tests whether **$file** does not exist. This test keeps **lnks** from wasting time looking for links to a nonexistent file. The test builtin, when called with the three arguments **!**, **–f**, and **$file**, evaluates to *true* if the file **$file** does *not* exist:

```
[ ! -f "$file" ]
```

The **!** operator preceding the **–f** argument to test negates its result, yielding *false* if the file **$file** *does* exist and is an ordinary file.

Next, **lnks** uses set and **ls –l** to check the number of links **$file** has:

```
# Check link count on file
set -- $(ls -l "$file")

linkcnt=$2
if [ "$linkcnt" -eq 1 ]; then
    echo "lnks: no other hard links to $file" 1>&2
    exit 0
fi
```

The set builtin uses command substitution (page 371) to set the positional parameters to the output of **ls –l**. The second field in this output is the link count, so the user-created variable **linkcnt** is set equal to **$2**. The **––** used with set prevents set from interpreting as an option the first argument produced by **ls –l** (the first argument is the access permissions for the file and typically begins with **–**). The **if** statement checks whether **$linkcnt** is equal to 1; if it is, **lnks** displays a message and exits. Although this message is not truly an error message, it is redirected to standard error. The way **lnks** has been written, all informational messages are sent to standard error. Only the final product of **lnks**—the pathnames of links to the specified file—is sent to standard output, so you can redirect the output.

If the link count is greater than 1, **lnks** goes on to identify the *inode* (page 1103) for **$file**. As explained on page 115, comparing the inodes associated with filenames is a good way to determine whether the filenames are links to the same file. The **lnks** script uses set to set the positional parameters to the output of **ls –i**. The first argument to set is the inode number for the file, so the user-created variable named **inode** is assigned the value of **$1**:

```
# Get the inode of the given file
set $(ls -i "$file")

inode=$1
```

Finally, **lnks** uses the find utility (page 822) to search for files having inode numbers that match **$inode**:

```
# Find and print the files with that inode number
echo "lnks: using find to search for links..." 1>&2
find "$directory" -xdev -inum $inode -print
```

The find utility searches the directory hierarchy rooted at the directory specified by its first argument (**$directory**) for files that meet the criteria specified by the remaining arguments. In this example, the remaining arguments send the names of files having inode numbers matching **$inode** to standard output. Because files in different filesystems can have the same inode number yet not be linked, find must search only directories in the same filesystem as **$directory**. The **–xdev** (cross-device) argument prevents find from searching directories on other filesystems. Refer to page 112 for more information about filesystems and links.

The echo command preceding the find command in **lnks**, which tells the user that find is running, is included because find can take a long time to run. Because **lnks** does not include a final exit statement, the exit status of **lnks** is that of the last command it runs: find.

Debugging Shell Scripts

When you are writing a script such as **lnks**, it is easy to make mistakes. You can use the shell's **–x** option to help debug a script. This option causes the shell to display each command after it expands it but before it runs the command. Tracing a script's execution in this way can give you information about where a problem lies.

You can run **lnks** (above) and cause the shell to display each command before it is executed. Either set the **–x** option for the current shell (**set –x**) so all scripts display commands as they are run or use the **–x** option to affect only the shell running the script called by the command line.

```
$ bash -x lnks letter /home
+ '[' 2 -eq 0 -o 2 -gt 2 ']'
+ '[' -d letter ']'
+ file=letter
+ '[' 2 -eq 1 ']'
+ '[' -d /home ']'
+ directory=/home
+ '[' '!' -f letter ']'
...
```

PS4 Each command the script executes is preceded by the value of the **PS4** variable—a plus sign (**+**) by default—so you can distinguish debugging output from output produced by the script. You must export **PS4** if you set it in the shell that calls the script. The next command sets **PS4** to **>>>>** followed by a SPACE and exports it:

```
$ export PS4='>>>> '
```

You can also set the **–x** option of the shell running the script by putting the following set command near the beginning of the script:

```
set -x
```

You can put **set –x** anywhere in the script to turn debugging on starting at that location. Turn debugging off using **set +x**. The **set –o xtrace** and **set +o xtrace** commands do the same things as **set –x** and **set +x**, respectively.

for...in

The **for...in** control structure (tcsh uses **foreach**) has the following syntax:

for ***loop-index*** *in* ***argument-list***
do
    commands
done

The **for...in** structure (Figure 10-4) assigns the value of the first argument in the ***argument-list*** to the ***loop-index*** and executes the ***commands*** between the **do** and **done** statements. The **do** and **done** statements mark the beginning and end of the **for** loop, respectively.

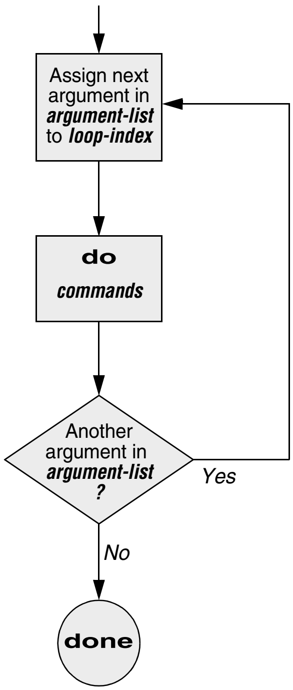

Figure 10-4 A **for...in** flowchart

After it passes control to the **done** statement, the structure assigns the value of the second argument in the ***argument-list*** to the ***loop-index*** and repeats the ***commands***. It repeats the ***commands*** between the **do** and **done** statements one time for each argument in the ***argument-list***. When the structure exhausts the ***argument-list***, it passes control to the statement following **done**.

The following **for...in** structure assigns **apples** to the user-created variable **fruit** and then displays the value of **fruit**, which is **apples**. Next, the structure assigns **oranges** to **fruit** and repeats the process. When it exhausts the argument list, the structure transfers control to the statement following **done**, which displays a message.

```
$ cat fruit
for fruit in apples oranges pears bananas
do
    echo "$fruit"
done
echo "Task complete."

$ ./fruit
apples
oranges
pears
bananas
Task complete.
```

The next script lists the names of the directory files in the working directory by looping through the files in the working directory and using test to determine which are directory files:

```
$ cat dirfiles
for i in *
do
    if [ -d "$i" ]
        then
            echo "$i"
    fi
done
```

The ambiguous file reference character * matches the names of all files (except hidden files) in the working directory. Prior to executing the **for** loop, the shell will expand the * and then uses the resulting list to assign successive values to the index variable **i**.

optional

Step Values

As an alternative to explicitly specifying values for ***argument-list***, you can specify step values. A **for...in** loop that uses step values assigns an initial value to or increments the ***loop-index***, executes the statements within the loop, and tests a termination condition at the end of the loop.

The following example uses brace expansion with a sequence expression (page 367) to generate the ***argument-list***. This syntax works on bash version 4.0 and above; give the command **echo $BASH_VERSION** to see which version you are using. The increment does not work under macOS. The first time through the loop, bash assigns a value of 0 to **count** (the ***loop-index***) and executes the statement between **do** and **done**. At the bottom of the loop, bash tests whether the termination condition has been met (is **count>10**?). If it has, bash passes control to the statement following **done**; if not, bash increments **count** by the increment value (**2**) and makes another pass through the loop. It repeats this process until the termination condition is met.

```
$ cat step1
for count in {0..10..2}
do
    echo -n "$count "
done
echo

$ ./step1
0 2 4 6 8 10
```

Older versions of bash do not support sequence expressions; you can use the seq utility to perform the same function:

```
$ for count in $(seq 0 2 10); do echo -n "$count "; done; echo
0 2 4 6 8 10
```

The next example uses bash's C-like syntax to specify step values. This syntax gives you more flexibility in specifying the termination condition and the increment value. Using this syntax, the first parameter initializes the ***loop-index***, the second parameter specifies the condition to be tested, and the third parameter specifies the increment.

```
$ cat rand
# $RANDOM evaluates to a random value 0 < x < 32,767
# This program simulates 10 rolls of a pair of dice
for ((x=1; x<=10; x++))
do
   echo -n "Roll #$x: "
   echo -n   $(( $RANDOM % 6 + 1 ))
   echo "  " $(( $RANDOM % 6 + 1 ))
done
```

for

The **for** control structure (not in tcsh) has the following syntax:

for ***loop-index***
do
    commands
done

In the **for** structure, the ***loop-index*** takes on the value of each of the command-line arguments, one at a time. The **for** structure is the same as the **for...in** structure (Figure 10-4, page 443) except in terms of where it gets values for the ***loop-index***. The **for** structure performs a sequence of commands, usually involving each argument in turn.

The following shell script shows a **for** structure displaying each command-line argument. The first line of the script, **for arg**, implies **for arg in "$@"**, where the shell expands **"$@"** into a list of quoted command-line arguments (i.e., **"$1" "$2" "$3"** ...). The balance of the script corresponds to the **for...in** structure.

```
$ cat for_test
for arg
do
    echo "$arg"
done

$ for_test candy gum chocolate
candy
gum
chocolate
```

The next example uses a different syntax. In it, the ***loop-index*** is named **count** and is set to an initial value of 0. The condition to be tested is **count<=10**: bash continues executing the loop as long as this condition is *true* (as long as **count** is less than or equal to 10; see Table 10-8 on page 508 for a list of operators). Each pass through the loop, bash adds 2 to the value of count (**count+=2**).

```
$ cat step2
for (( count=0; count<=10; count+=2 ))
do
    echo -n "$count "
done
echo

$ ./step2
0 2 4 6 8 10
```

optional

THE whos SCRIPT

The following script, named **whos**, demonstrates the usefulness of the implied **"$@"** in the **for** structure. You give **whos** one or more users' full names or usernames as arguments, and **whos** displays information about the users. The **whos** script gets the information it displays from the first and fifth fields in the **/etc/passwd** file. The first field contains a username, and the fifth field typically contains the user's full name. You can provide a username as an argument to **whos** to display the user's name or provide a name as an argument to display the username. The **whos** script is similar to the finger utility, although **whos** delivers less information. macOS uses Open Directory in place of the **passwd** file; see page 1068 for a similar script that runs under macOS.

```
$ cat whos
#!/bin/bash

if [ $# -eq 0 ]
    then
        echo "Usage: whos id..." 1>&2
        exit 1
fi
```

```
for id
do
    gawk -F: '{print $1, $5}' /etc/passwd |
    grep -i "$id"
done
```

In the next example, **whos** identifies the user whose username is **chas** and the user whose name is **Marilou Smith**:

```
$ ./whos chas "Marilou Smith"
chas Charles Casey
msmith Marilou Smith
```

Use of **"$@"** The **whos** script uses a **for** statement to loop through the command-line arguments. In this script the implied use of **"$@"** in the **for** loop is particularly beneficial because it causes the **for** loop to treat an argument that contains a SPACE as a single argument. This example encloses **Marilou Smith** in quotation marks, which causes the shell to pass it to the script as a single argument. Then the implied **"$@"** in the **for** statement causes the shell to regenerate the quoted argument **Marilou Smith** so that it is again treated as a single argument. The double quotation marks in the grep statement perform the same function.

gawk For each command-line argument, **whos** searches the **/etc/passwd** file. Inside the **for** loop, the gawk utility (Chapter 14; awk and mawk work the same way) extracts the first (**$1**) and fifth (**$5**) fields from each line in **/etc/passwd**. The **–F:** option causes gawk to use a colon (**:**) as a field separator when it reads **/etc/passwd**, allowing it to break each line into fields. The gawk command sets and uses the **$1** and **$5** arguments; they are included within single quotation marks and are not interpreted by the shell. Do not confuse these arguments with positional parameters, which will correspond to command-line arguments. The first and fifth fields are sent to grep (page 853) via a pipeline. The grep utility searches for **$id** (to which the shell has assigned the value of a command-line argument) in its input. The **–i** option causes grep to ignore case as it searches; grep displays each line in its input that contains **$id**.

A pipe symbol (|) at the end of a line Under bash (and not tcsh), a control operator such as a pipe symbol (|) implies continuation: bash "knows" another command must follow it. Therefore, in **whos**, the NEWLINE following the pipe symbol at the end of the line with the gawk command does not have to be quoted. For more information refer to "Implicit Command-Line Continuation" on page 512.

while

The **while** control structure (see page 416 for tcsh) has the following syntax:

while ***test-command***
do
    commands
done

As long as the ***test-command*** (Figure 10-5) returns a *true* exit status, the **while** structure continues to execute the series of ***commands*** delimited by the **do** and **done** statements. Before each loop through the ***commands***, the structure executes the ***test-command***. When the exit status of the ***test-command*** is *false*, the structure passes control to the statement after the **done** statement.

test builtin The following shell script first initializes the **number** variable to zero. The test builtin then determines whether **number** is less than 10. The script uses test with the **–lt** argument to perform a numerical test. For numerical comparisons, you must use **–ne** (not equal), **–eq** (equal), **–gt** (greater than), **–ge** (greater than or equal to), **–lt** (less than), or **–le** (less than or equal to). For string comparisons, use **=** (equal) or **!=** (not equal) when you are working with test. In this example, test has an exit status of 0 (*true*) as long as **number** is less than 10. As long as test returns *true*, the structure executes the commands between the **do** and **done** statements. See page 1005 for information on the test utility, which is very similar to the test builtin.

```
$ cat count
#!/bin/bash
number=0
while [ "$number" -lt 10 ]
    do
        echo -n "$number"
        ((number +=1))
    done
echo
$ ./count
0123456789
$
```

The echo command following **do** displays **number**. The **–n** prevents echo from issuing a NEWLINE following its output. The next command uses arithmetic evaluation [((**...**)); page 505] to increment the value of **number** by 1. The **done** statement terminates the loop and returns control to the **while** statement to start the loop over again. The final echo causes **count** to send a NEWLINE character to standard output, so the next prompt is displayed at the left edge of the display rather than immediately following the 9.

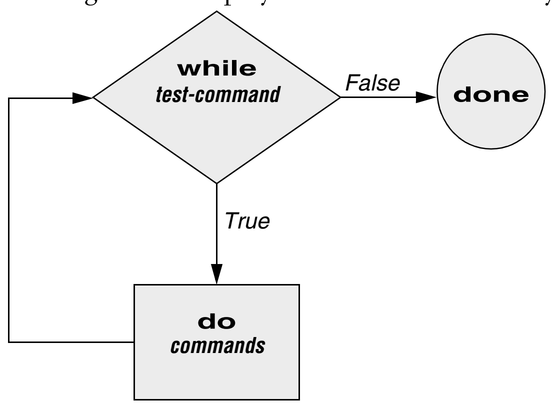

Figure 10-5 A **while** flowchart

optional The spell_check Script

The aspell utility (page 739; not available under macOS) checks the words in a file against a dictionary of correctly spelled words. With the **list** command, aspell runs in list mode: Input comes from standard input and aspell sends each potentially misspelled word to standard output. The following command produces a list of possible misspellings in the file **letter.txt**:

```
$ aspell list < letter.txt
quikly
portible
frendly
```

The next shell script, named **spell_check**, shows another use of a **while** structure. To find the incorrect spellings in a file, **spell_check** calls aspell to check a file against a system dictionary. But it goes a step further: It enables you to specify a list of correctly spelled words and removes these words from the output of aspell. This script is useful for removing words you use frequently, such as names and technical terms, that do not appear in a standard dictionary. Although you can duplicate the functionality of **spell_check** by using additional aspell dictionaries, the script is included here for its instructive value.

The **spell_check** script requires two filename arguments: the file containing the list of correctly spelled words and the file you want to check. The first **if** statement verifies that the user specified two arguments. The next two **if** statements verify that both arguments are readable files. (The exclamation point negates the sense of the following operator; the **–r** operator causes test to determine whether a file is readable. The result is a test that determines whether a file is *not readable*.)

```
$ cat spell_check
#!/bin/bash
# remove correct spellings from aspell output

if [ $# -ne 2 ]
    then
        echo "Usage: spell_check dictionary filename" 1>&2
        echo "dictionary: list of correct spellings" 1>&2
        echo "filename: file to be checked" 1>&2
        exit 1
fi

if [ ! -r "$1" ]
    then
        echo "spell_check: $1 is not readable" 1>&2
        exit 1
fi
if [ ! -r "$2" ]
    then
        echo "spell_check: $2 is not readable" 1>&2
        exit 1
fi
```

```
aspell list < "$2" |
while read line
do
    if ! grep "^$line$" "$1" > /dev/null
        then
            echo $line
    fi
done
```

The **spell_check** script sends the output from aspell (with the **list** argument, so it produces a list of misspelled words on standard output) through a pipeline to standard input of a **while** structure, which reads one line at a time (each line has one word on it) from standard input. The ***test-command*** (that is, **read line**) returns a *true* exit status as long as it receives a line from standard input.

Inside the **while** loop, an **if** statement monitors the return value of grep, which determines whether the line that was read is in the user's list of correctly spelled words. The pattern grep searches for (the value of **$line**) is preceded and followed by special characters that specify the beginning and end of a line (**^** and **$**, respectively). These special characters ensure that grep finds a match only if the **$line** variable matches an entire line in the file of correctly spelled words. (Otherwise, grep would match a string, such as **paul**, in the output of aspell if the file of correctly spelled words contained the word **paulson**.) These special characters, together with the value of the **$line** variable, form a regular expression (Appendix A).

The output of grep is redirected to **/dev/null** (page 145) because the output is not needed; only the exit code is important. The **if** statement checks the negated exit status of grep (the leading exclamation point negates or changes the sense of the exit status—*true* becomes *false,* and vice versa), which is 0 or *true* (*false* when negated) when a matching line is found. If the exit status is *not* 0 or *false* (*true* when negated), the word was *not* in the file of correctly spelled words. The echo builtin sends a list of words that are not in the file of correctly spelled words to standard output.

Once it detects the EOF (end of file), the read builtin returns a *false* exit status, control passes out of the **while** structure, and the script terminates.

Before you use **spell_check**, create a file of correct spellings containing words that you use frequently but that are not in a standard dictionary. For example, if you work for a company named **Blinkenship and Klimowski, Attorneys**, you would put **Blinkenship** and **Klimowski** in the file. The following example shows how **spell_check** checks the spelling in a file named **memo** and removes **Blinkenship** and **Klimowski** from the output list of incorrectly spelled words:

```
$ aspell list < memo
Blinkenship
Klimowski
targat
hte
$ cat word_list
Blinkenship
Klimowski
$ ./spell_check word_list memo
targat
hte
```

until

The **until** (not in tcsh) and **while** (see page 416 for tcsh) structures are similar, differing only in the sense of the test performed at the top of the loop. Figure 10-6 shows that **until** continues to loop *until* the ***test-command*** returns a *true* exit status. The **while** structure loops *while* the ***test-command*** continues to return a *true* or nonerror condition. The **until** control structure has the following syntax:

until ***test-command***
do
    commands
done

The following script demonstrates an **until** structure that includes read (page 489). When the user enters the correct string of characters, the ***test-command*** is satisfied and the structure passes control out of the loop.

```
$ cat until1
secretname=zach
name=noname
echo "Try to guess the secret name!"
echo
until [ "$name" = "$secretname" ]
do
    read -p "Your guess: " name
done
echo "Very good."

$ ./until1
Try to guess the secret name!

Your guess: helen
Your guess: barbara
Your guess: rachael
Your guess: zach
Very good
```

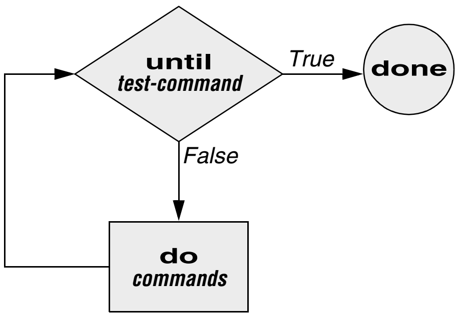

Figure 10-6 An **until** flowchart

The following **locktty** script is similar to the lock command on Berkeley UNIX and the **Lock Screen** menu selection in GNOME. The script prompts for a key (password) and uses an **until** control structure to lock the terminal. The **until** statement causes the system to ignore any characters typed at the keyboard until the user types the key followed by a RETURN on a line by itself, which unlocks the terminal. The **locktty** script can keep people from using your terminal while you are away from it for short periods of time. It saves you from having to log out if you are concerned about other users using your session.

```
$ cat locktty
#! /bin/bash

trap '' 1 2 3 18
stty -echo
read -p "Key: " key_1
echo
read -p "Again: " key_2
echo
key_3=
if [ "$key_1" = "$key_2" ]
    then
        tput clear
        until [ "$key_3" = "$key_2" ]
        do
            read key_3
        done
    else
        echo "locktty: keys do not match" 1>&2
fi
stty echo
```

Forget your password for locktty?

tip If you forget your key (password), you will need to log in from another (virtual) terminal and give a command to kill the process running **locktty** (e.g., **killall –9 locktty**).

trap builtin The trap builtin (page 496; not in tcsh) at the beginning of the **locktty** script stops a user from being able to terminate the script by sending it a signal (for example, by pressing the interrupt key). Trapping signal 20 means that no one can use CONTROL-Z (job control, a stop from a tty) to defeat the lock. Table 10-5 on page 496 provides a list of signals. The **stty –echo** command (page 987) turns on keyboard echo (causes the terminal not to display characters typed at the keyboard), preventing the key the user enters from appearing on the screen. After turning off keyboard echo, the script prompts the user for a key, reads it into the user-created variable **key_1**, prompts the user to enter the same key again, and saves it in **key_2**. The statement **key_3=** creates a variable with a NULL value. If **key_1** and **key_2** match, **locktty** clears the screen (with the tput command) and starts an **until** loop. The **until** loop keeps reading from the terminal and assigning the input to the **key_3** variable. Once the user types a string that matches one of the original keys (**key_2**), the **until** loop terminates and keyboard echo is turned on again.

break AND continue

You can interrupt a **for**, **while**, or **until** loop by using a **break** or **continue** statement. The **break** statement transfers control to the statement following the **done** statement, thereby terminating execution of the loop. The **continue** command transfers control to the **done** statement, continuing execution of the loop.

The following script demonstrates the use of these two statements. The **for...in** structure loops through the values 1–10. The first **if** statement executes its commands when the value of the index is less than or equal to 3 (**$index –le 3**). The second **if** statement executes its commands when the value of the index is greater than or equal to 8 (**$index –ge 8**). In between the two **if**s, echo displays the value of the index. For all values up to and including 3, the first **if** statement displays **continue**, executes a **continue** statement that skips **echo $index** and the second **if** statement, and continues with the next **for** statement. For the value of 8, the second **if** statement displays the word **break** and executes a **break** statement that exits from the **for** loop.

```
$ cat brk
for index in 1 2 3 4 5 6 7 8 9 10
    do
        if [ $index -le 3 ] ; then
            echo "continue"
            continue
        fi
#
    echo $index
#
    if [ $index -ge 8 ] ; then
        echo "break"
        break
    fi
done

$ ./brk
continue
continue
continue
4
5
6
7
8
break
$
```

case

The **case** structure (Figure 10-7; tcsh uses **switch**) is a multiple-branch decision mechanism. The path taken through the structure depends on a match or lack of a match between the ***test-string*** and one of the ***patterns***. When the ***test-string*** matches one of the ***patterns***, the shell transfers control to the ***commands*** following the ***pattern***. The ***commands*** are terminated by a double semicolon (**;;**) control operator. When control reaches this control operator, the shell transfers control to the command following the **esac** statement. The **case** control structure has the following syntax:

case ***test-string*** *in*
    pattern-1*)*
        commands-1
        ;;

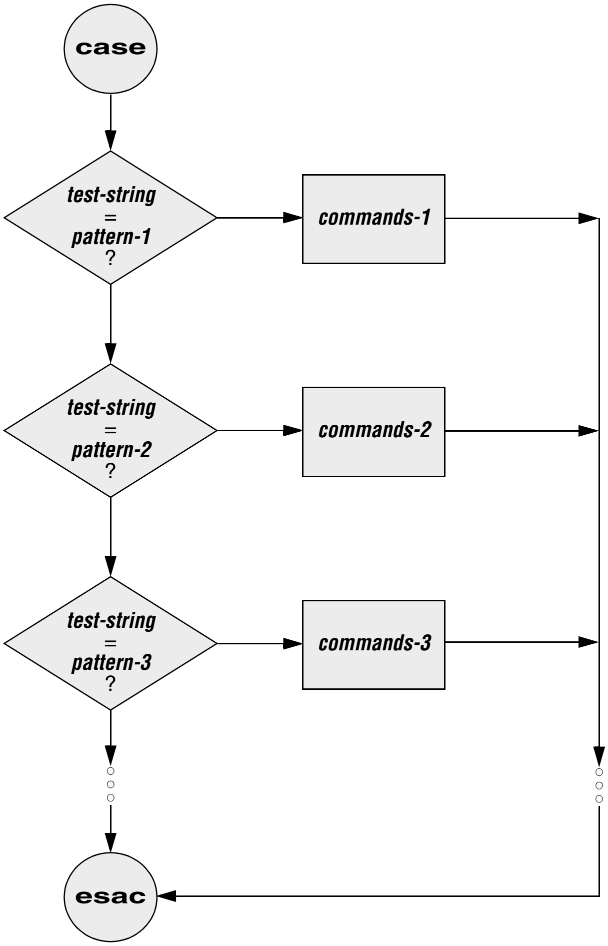

Figure 10-7 A **case** flowchart

```
        pattern-2)
            commands-2
            ;;
        pattern-3)
            commands-3
            ;;
    . . .
    esac
```

The following **case** structure uses the character the user enters as the ***test-string***. This value is held in the variable **letter**. If the ***test-string*** has a value of **A**, the structure executes the command following the ***pattern*** **A**. The right parenthesis is part of the **case** control structure, not part of the ***pattern***. If the ***test-string*** has a value of **B** or **C**, the structure executes the command following the matching ***pattern***. The asterisk (*) indicates *any string of characters* and serves as a catchall in case there is no match. If no ***pattern*** matches the ***test-string*** and if there is no catchall (*) ***pattern***, control passes to the command following the **esac** statement, without the **case** structure taking any action.

```
$ cat case1
read -p "Enter A, B, or C: " letter
case "$letter" in
    A)
        echo "You entered A"
        ;;
    B)
        echo "You entered B"
        ;;
    C)
        echo "You entered C"
        ;;
    *)
        echo "You did not enter A, B, or C"
        ;;
esac

$ ./case1
Enter A, B, or C: B
You entered B
```

The next execution of **case1** shows the user entering a lowercase **b**. Because the ***test-string*** **b** does not match the uppercase **B** ***pattern*** (or any other ***pattern*** in the **case** statement), the program executes the commands following the catchall ***pattern*** and displays a message:

```
$ ./case1
Enter A, B, or C: b
You did not enter A, B, or C
```

The *pattern* in the **case** structure is a glob (it is analogous to an ambiguous file reference). It can include any special characters and strings shown in Table 10-2.

Table 10-2 Patterns

Pattern	Function
*	Matches any string of characters. Use for the default case.
?	Matches any single character.
[...]	Defines a character class. Any characters enclosed within brackets are tried, one at a time, in an attempt to match a single character. A hyphen between two characters specifies a range of characters.
|	Separates alternative choices that satisfy a particular branch of the **case** structure.

The next script accepts both uppercase and lowercase letters:

```
$ cat case2
read -p "Enter A, B, or C: " letter
case "$letter" in
    a|A)
        echo "You entered A"
        ;;
    b|B)
        echo "You entered B"
        ;;
    c|C)
        echo "You entered C"
        ;;
    *)
        echo "You did not enter A, B, or C"
        ;;
esac

$ ./case2
Enter A, B, or C: b
You entered B
```

optional The following example shows how to use the **case** structure to create a simple menu. The **command_menu** script uses echo to present menu items and prompt the user for a selection. (The **select** control structure [page 460] is a much easier way of coding a menu.) The **case** structure then executes the appropriate utility depending on the user's selection.

```
$ cat command_menu
#!/bin/bash
# menu interface to simple commands

echo -e "\n      COMMAND MENU\n"
echo "  a.  Current date and time"
echo "  b.  Users currently logged in"
```

```
echo "  c.  Name of the working directory"
echo -e "  d.  Contents of the working directory\n"
read -p "Enter a, b, c, or d: " answer
echo
#
case "$answer" in
    a)
        date
        ;;
    b)
        who
        ;;
    c)
        pwd
        ;;
    d)
        ls
        ;;
    *)
        echo "There is no selection: $answer"
        ;;
esac

$ ./command_menu

            COMMAND MENU

    a.  Current date and time
    b.  Users currently logged in
    c.  Name of the working directory
    d.  Contents of the working directory

Enter a, b, c, or d: a
Sat Jan  6 12:31:12 PST 2018
```

echo –e The **–e** option causes echo to interpret **\n** as a NEWLINE character. If you do not include this option, echo does not output the extra blank lines that make the menu easy to read but instead outputs the (literal) two-character sequence **\n**. The **–e** option causes echo to interpret several other backslash-quoted characters (Table 10-3). Remember to quote (i.e., place double quotation marks around the string) the backslash-quoted character so the shell does not interpret it but rather passes the backslash and the character to echo. See **xpg_echo** (page 363) for a way to avoid using the **–e** option.

Table 10-3 Special characters in echo (must use **–e**)

Quoted character	echo displays
\a	Alert (bell)
\b	BACKSPACE
\c	Suppress trailing NEWLINE

Table 10-3 Special characters in echo (must use **–e**) (continued)

Quoted character	echo **displays**
\f	FORMFEED
\n	NEWLINE
\r	RETURN
\t	Horizontal TAB
\v	Vertical TAB
****	Backslash
nnn	The character with the ASCII octal code ***nnn;*** if ***nnn*** is not valid, echo displays the string literally

You can also use the **case** control structure to take various actions in a script, depending on how many arguments the script is called with. The following script, named **safedit**, uses a **case** structure that branches based on the number of command-line arguments (**$#**). It calls vim and saves a backup copy of a file you are editing.

```
$ cat safedit
#!/bin/bash

PATH=/bin:/usr/bin
script=$(basename $0)
case $# in

    0)
        vim
        exit 0
        ;;

    1)
        if [ ! -f "$1" ]
            then
                vim "$1"
                exit 0
            fi
        if [ ! -r "$1" -o ! -w "$1" ]
            then
                echo "$script: check permissions on $1" 1>&2
                exit 1
            else
                editfile=$1
            fi
        if [ ! -w "." ]
            then
                echo "$script: backup cannot be " \
                    "created in the working directory" 1>&2
                exit 1
            fi
        ;;
```

```
        *)
            echo "Usage: $script [file-to-edit]" 1>&2
            exit 1
            ;;
esac
tempfile=/tmp/$$.$script
cp $editfile $tempfile
if vim $editfile
    then
        mv $tempfile bak.$(basename $editfile)
        echo "$script: backup file created"
    else
        mv $tempfile editerr
        echo "$script: edit error--copy of " \
            "original file is in editerr" 1>&2
fi
```

If you call **safedit** without any arguments, the **case** structure executes its first branch and calls vim without a filename argument. Because an existing file is not being edited, **safedit** does not create a backup file. (See the **:w** command on page 179 for an explanation of how to exit from vim when you have called it without a filename.) If you call **safedit** with one argument, it runs the commands in the second branch of the **case** structure and verifies that the file specified by **$1** does not yet exist or is the name of a file for which the user has read and write permission. The **safedit** script also verifies that the user has write permission for the working directory. If the user calls **safedit** with more than one argument, the third branch of the **case** structure presents a usage message and exits with a status of 1.

Set **PATH** At the beginning of the script, the **PATH** variable is set to search **/bin** and **/usr/bin**. Setting **PATH** in this way ensures that the commands executed by the script are standard utilities, which are kept in those directories. By setting this variable inside a script, you can avoid the problems that might occur if users have set **PATH** to search their own directories first and have scripts or programs with the same names as the utilities the script calls. You can also include absolute pathnames within a script to achieve this end, although this practice can make a script less portable.

Name of the program The next line declares a variable named **script** and initializes it with the simple filename of the script:

```
script=$(basename $0)
```

The basename utility sends the simple filename component of its argument to standard output, which is assigned to the **script** variable, using command substitution. The **$0** holds the command the script was called with (page 470). No matter which of the following commands the user calls the script with, the output of basename is the simple filename **safedit**:

```
$ /home/max/bin/safedit memo
$ ./safedit memo
$ safedit memo
```

After the **script** variable is set, it replaces the filename of the script in usage and error messages. By using a variable that is derived from the command that invoked the script rather than a filename that is hardcoded into the script, you can create links to the script or rename it, and the usage and error messages will still provide accurate information.

Naming temporary files

Another feature of **safedit** relates to the use of the **$$** parameter in the name of a temporary file. The statement following the **esac** statement creates and assigns a value to the **tempfile** variable. This variable contains the name of a temporary file that is stored in the **/tmp** directory, as are many temporary files. The temporary filename begins with the PID number of the shell and ends with the name of the script. Using the PID number ensures that the filename is unique. Thus **safedit** will not attempt to overwrite an existing file, as might happen if two people were using **safedit** at the same time. The name of the script is appended so that, should the file be left in **/tmp** for some reason, you can figure out where it came from.

The PID number is used in front of—rather than after—**$script** in the filename because of the 14-character limit placed on filenames by some older versions of UNIX. Linux systems do not have this limitation. Because the PID number ensures the uniqueness of the filename, it is placed first so that it cannot be truncated. (If the **$script** component is truncated, the filename is still unique.) For the same reason, when a backup file is created inside the **if** control structure a few lines down in the script, the filename consists of the string **bak.** followed by the name of the file being edited. On an older system, if **bak** were used as a suffix rather than a prefix and the original filename were 14 characters long, **.bak** might be lost and the original file would be overwritten. The basename utility extracts the simple filename of **$editfile** before it is prefixed with **bak.**.

The **safedit** script uses an unusual ***test-command*** in the **if** structure: **vim $editfile**. The ***test-command*** calls vim to edit **$editfile**. When you finish editing the file and exit from vim, vim returns an exit code. The **if** control structure uses that exit code to determine which branch to take. If the editing session completed successfully, vim returns **0** and the statements following the **then** statement are executed. If vim does not terminate normally (as would occur if the user killed [page 866] the vim process), vim returns a nonzero exit status and the script executes the statements following **else**.

select

The **select** control structure (not in tcsh) is based on the one found in the Korn Shell. It displays a menu, assigns a value to a variable based on the user's choice of items, and executes a series of commands. The **select** control structure has the following syntax:

select ***varname*** *[in* ***arg*** *. . .]*
do
  commands
done

The **select** structure displays a menu of the ***arg*** items. If you omit the keyword **in** and the list of arguments, **select** uses the positional parameters in place of the ***arg*** items. The menu is formatted with numbers before each item. For example, a **select** structure that begins with

```
select fruit in apple banana blueberry kiwi orange watermelon STOP
```

displays the following menu:

```
1) apple      3) blueberry   5) orange      7) STOP
2) banana     4) kiwi        6) watermelon
```

The **select** structure uses the values of the **LINES** (default is 24) and **COLUMNS** (default is 80) variables to specify the size of the display. With **COLUMNS** set to 20, the menu looks like this:

```
1) apple
2) banana
3) blueberry
4) kiwi
5) orange
6) watermelon
7) STOP
```

PS3 After displaying the menu, **select** displays the value of **PS3**, the **select** prompt. The default value of **PS3** is **?#**, but it is typically set to a more meaningful value. When you enter a valid number (one in the menu range) in response to the **PS3** prompt, **select** sets ***varname*** to the argument corresponding to the number you entered. An invalid entry causes the shell to set ***varname*** to null. Either way, **select** stores your response in the keyword variable **REPLY** and then executes the ***commands*** between **do** and **done**. If you press RETURN without entering a choice, the shell redisplays the menu and the **PS3** prompt.

The **select** structure continues to issue the **PS3** prompt and execute the ***commands*** until something causes it to exit—typically, a **break** or **exit** statement. A **break** statement exits from the loop and an **exit** statement exits from the script.

The following script illustrates the use of **select**:

```
$ cat fruit2
#!/bin/bash
PS3="Choose your favorite fruit from these possibilities: "
select FRUIT in apple banana blueberry kiwi orange watermelon STOP
do
    if [ "$FRUIT" == "" ]; then
        echo -e "Invalid entry.\n"
        continue
    elif [ $FRUIT = STOP ]; then
        echo "Thanks for playing!"
        break
    fi
echo "You chose $FRUIT as your favorite."
echo -e "That is choice number $REPLY.\n"
done
```

```
$ ./fruit2
1) apple      3) blueberry   5) orange      7) STOP
2) banana     4) kiwi        6) watermelon
Choose your favorite fruit from these possibilities: 3
You chose blueberry as your favorite.
That is choice number 3.

Choose your favorite fruit from these possibilities: 99
Invalid entry.

Choose your favorite fruit from these possibilities: 7
Thanks for playing!
```

After setting the **PS3** prompt and establishing the menu with the **select** statement, **fruit2** executes the ***commands*** between **do** and **done**. If the user submits an invalid entry, the shell sets ***varname*** (**$FRUIT**) to a null value. If **$FRUIT** is null, echo displays an error message; **continue** then causes the shell to redisplay the **PS3** prompt. If the entry is valid, the script tests whether the user wants to stop. If so, echo displays an appropriate message and **break** exits from the **select** structure (and from the script). If the user enters a valid response and does not want to stop, the script displays the name and number of the user's response. (See page 457 for information about the echo –e option.)

Here Document

A Here document allows you to redirect input to a shell script from within the shell script itself. A Here document is so named because it is *here*—immediately accessible in the shell script—instead of *there,* perhaps in another file.

The following script, named **birthday**, contains a Here document. The two less than symbols (**<<**) in the first line indicate a Here document follows. One or more characters that delimit the Here document follow the less than symbols—this example uses a plus sign. Whereas the opening delimiter must appear adjacent to the less than symbols, the closing delimiter must be on a line by itself. The shell sends everything between the two delimiters to the process as standard input. In the example it is as though you have redirected standard input to grep from a file, except that the file is embedded in the shell script:

```
$ cat birthday
grep -i "$1" <<+
Max     June 22
Barbara February 3
Darlene May 8
Helen   March 13
Zach    January 23
Nancy   June 26
+
$ ./birthday Zach
Zach    January 23
```

```
$ ./birthday june
Max     June 22
Nancy   June 26
```

When you run **birthday**, it lists all the Here document lines that contain the argument you called it with. In this case the first time **birthday** is run, it displays Zach's birthday because it is called with an argument of **Zach**. The second run displays all the birthdays in June. The **–i** argument causes grep's search not to be case sensitive.

optional The next script, named **bundle**,[1] includes a clever use of a Here document. The **bundle** script is an elegant example of a script that creates a shell archive (**shar**) file. The script creates a file that is itself a shell script containing several other files as well as the code needed to re-create the original files:

```
$ cat bundle
#!/bin/bash
# bundle:  group files into distribution package

echo "# To unbundle, bash this file"
for i
do
    echo "echo $i 1>&2"
    echo "cat >$i <<'End of $i'"
    cat $i
    echo "End of $i"
done
```

Just as the shell does not treat special characters that occur in standard input of a shell script as special, so the shell does not treat the special characters that occur between the delimiters in a Here document as special.

As the following example shows, the output of **bundle** is a shell script, which is redirected to a file named **bothfiles**. It contains the contents of each file given as an argument to **bundle** (**file1** and **file2** in this case) inside a Here document. To extract the original files from **bothfiles**, you simply give it as an argument to a bash command. Before each Here document is a cat command that causes the Here document to be written to a new file when **bothfiles** is run:

```
$ cat file1
This is a file.
It contains two lines.
$ cat file2
This is another file.
It contains
three lines.
```

1. Thanks to Brian W. Kernighan and Rob Pike, *The Unix Programming Environment* (Englewood Cliffs, N.J.: Prentice-Hall, 1984), 98. Reprinted with permission.

```
$ ./bundle file1 file2 > bothfiles
$ cat bothfiles
# To unbundle, bash this file
echo file1 1>&2
cat >file1 <<'End of file1'
This is a file.
It contains two lines.
End of file1
echo file2 1>&2
cat >file2 <<'End of file2'
This is another file.
It contains
three lines.
End of file2
```

In the next example, **file1** and **file2** are removed before **bothfiles** is run. The **bothfiles** script echoes the names of the files it creates as it creates them. The ls command then shows that **bothfiles** has re-created **file1** and **file2**:

```
$ rm file1 file2
$ bash bothfiles
file1
file2
$ ls
bothfiles
file1
file2
```

File Descriptors

As discussed on page 292, before a process can read from or write to a file, it must open that file. When a process opens a file, Linux associates a number (called a *file descriptor*) with the file. A file descriptor is an index into the process's table of open files. Each process has its own set of open files and its own file descriptors. After opening a file, a process reads from and writes to that file by referring to its file descriptor. When it no longer needs the file, the process closes the file, freeing the file descriptor.

A typical Linux process starts with three open files: standard input (file descriptor 0), standard output (file descriptor 1), and standard error (file descriptor 2). Often, these are the only files the process needs. Recall that you redirect standard output with the symbol **>** or the symbol **1>** and that you redirect standard error with the symbol **2>**. Although you can redirect other file descriptors, because file descriptors other than 0, 1, and 2 do not have any special conventional meaning, it is rarely useful to do so. The exception is in programs that you write yourself, in which case you control the meaning of the file descriptors and can take advantage of redirection.

Opening a File Descriptor

The Bourne Again Shell opens files using the exec builtin with the following syntax:

exec ***n*****>** ***outfile***
exec ***m*****<** ***infile***

The first line opens ***outfile*** for output and holds it open, associating it with file descriptor ***n***. The second line opens ***infile*** for input and holds it open, associating it with file descriptor ***m***.

Duplicating a File Descriptor

The **<&** token duplicates an input file descriptor; **>&** duplicates an output file descriptor. You can duplicate a file descriptor by making it refer to the same file as another open file descriptor, such as standard input or output. Use the following syntax to open or redirect file descriptor ***n*** as a duplicate of file descriptor ***m:***

exec ***n*****<&*****m***

Once you have opened a file, you can use it for input and output in two ways. First, you can use I/O redirection on any command line, redirecting standard output to a file descriptor with **>&*****n*** or redirecting standard input from a file descriptor with **<&*****n***. Second, you can use the read (page 489) and echo builtins. If you invoke other commands, including functions (page 356), they inherit these open files and file descriptors. When you have finished using a file, you can close it using the following syntax:

exec ***n*****<&–**

File Descriptor Examples

When you call the following **mycp** function with two arguments, it copies the file named by the first argument to the file named by the second argument. If you supply only one argument, the script copies the file named by the argument to standard output. If you invoke **mycp** with no arguments, it copies standard input to standard output.

A function is not a shell script

tip The **mycp** example is a shell function; it will not work as you expect if you execute it as a shell script. (It *will* work: The function will be created in a very short-lived subshell, which is of little use.) You can enter this function from the keyboard. If you put the function in a file, you can run it as an argument to the **.** (dot) builtin (page 290). You can also put the function in a startup file if you want it to be always available (page 358).

```
function mycp () {
case $# in
    0)
        # Zero arguments
        # File descriptor 3 duplicates standard input
        # File descriptor 4 duplicates standard output
        exec 3<&0 4<&1
        ;;
    1)
        # One argument
        # Open the file named by the argument for input
        # and associate it with file descriptor 3
        # File descriptor 4 duplicates standard output
        exec 3< $1 4<&1
        ;;
    2)
        # Two arguments
        # Open the file named by the first argument for input
        # and associate it with file descriptor 3
        # Open the file named by the second argument for output
        # and associate it with file descriptor 4
        exec 3< $1 4> $2
        ;;
    *)
        echo "Usage: mycp [source [dest]]"
        return 1
        ;;
esac

# Call cat with input coming from file descriptor 3
# and output going to file descriptor 4
cat <&3 >&4

# Close file descriptors 3 and 4
exec 3<&- 4<&-
}
```

The real work of this function is done in the line that begins with cat. The rest of the script arranges for file descriptors 3 and 4, which are the input and output of the cat command, respectively, to be associated with the appropriate files.

optional The next program takes two filenames on the command line, sorts both, and sends the output to temporary files. The program then merges the sorted files to standard output, preceding each line with a number that indicates which file it came from.

```
$ cat sortmerg
#!/bin/bash
usage () {
if [ $# -ne 2 ]; then
    echo "Usage: $0 file1 file2" 2>&1
    exit 1
    fi
}

# Default temporary directory
: ${TEMPDIR:=/tmp}
```

```
# Check argument count
usage "$@"

# Set up temporary files for sorting
file1=$TEMPDIR/$$.file1
file2=$TEMPDIR/$$.file2

# Sort
sort $1 > $file1
sort $2 > $file2

# Open $file1 and $file2 for reading. Use file descriptors 3 and 4.
exec 3<$file1
exec 4<$file2

# Read the first line from each file to figure out how to start.
read Line1 <&3
status1=$?
read Line2 <&4
status2=$?
# Strategy: while there is still input left in both files:
#   Output the line that should come first.
#   Read a new line from the file that line came from.
while [ $status1 -eq 0 -a $status2 -eq 0 ]
    do
        if [[ "$Line2" > "$Line1" ]]; then
            echo -e "1.\t$Line1"
            read -u3 Line1
            status1=$?
        else
            echo -e "2.\t$Line2"
            read -u4 Line2
            status2=$?
        fi
    done

# Now one of the files is at end of file.
# Read from each file until the end.
# First file1:
while [ $status1 -eq 0 ]
    do
        echo -e "1.\t$Line1"
        read Line1 <&3
        status1=$?
    done
# Next file2:
while [[ $status2 -eq 0 ]]
    do
        echo -e "2.\t$Line2"
        read Line2 <&4
        status2=$?
    done

# Close and remove both input files
exec 3<&- 4<&-
rm -f $file1 $file2
exit 0
```

Determining Whether a File Descriptor Is Associated with the Terminal

The test –t criterion takes an argument of a file descriptor and causes test to return a value of 0 (*true*) or not 0 (*false*) based on whether the specified file descriptor is associated with the terminal (screen or keyboard). It is typically used to determine whether standard input, standard output, and/or standard error is coming from/going to the terminal.

In the following example, the **is.term** script uses the test –t criterion ([ ] is a synonym for test; page 1005) to see if file descriptor 1 (initially standard output) of the process running the shell script is associated with the screen. The message the script displays is based on whether test returns *true* (file descriptor 1 is associated with the screen) or *false* (file descriptor 1 is *not* associated with the screen).

```
$ cat is.term
if [ -t 1 ] ; then
        echo "FD 1 (stdout) IS going to the screen"
    else
        echo "FD 1 (stdout) is NOT going to the screen"
fi
```

When you run **is.term** without redirecting standard output, the script displays **FD 1 (stdout) IS going to the screen** because standard output of the **is.term** script has not been redirected:

```
$ ./is.term
FD 1 (stdout) IS going to the screen
```

When you redirect standard output of a program using **>** on the command line, bash closes file descriptor 1 and then reopens it, associating it with the file specified following the redirect symbol.

The next example redirects standard output of the **is.term** script: The newly opened file descriptor 1 associates standard output with the file named **hold**. Now the test command ([**–t 1**]) fails, returning a value of 1 (*false*), because standard output is not associated with a terminal. The script writes **FD 1 (stdout) is NOT going to the screen** to **hold**:

```
$ ./is.term > hold
$ cat hold
FD 1 (stdout) is NOT going to the screen
```

If you redirect standard error from **is.term**, the script will report **FD 1 (stdout) IS going to the screen** and will write nothing to the file receiving the redirection; standard output has not been redirected. You can use [**–t 2**] to test if standard error is going to the screen:

```
$ ./is.term 2> hold
FD 1 (stdout) IS going to the screen
```

In a similar manner, if you send standard output of **is.term** through a pipeline, test reports standard output is not associated with a terminal. In this example, cat copies standard input to standard output:

```
$ ./is.term | cat
FD 1 (stdout) is NOT going to the screen
```

optional You can also experiment with test on the command line. This technique allows you to make changes to your experimental code quickly by taking advantage of command history and editing (page 338). To better understand the following examples, first verify that test (called as [ ]) returns a value of 0 (*true*) when file descriptor 1 is associated with the screen and a value other than 0 (*false*) when file descriptor 1 is not associated with the screen. The **$?** special parameter (page 477) holds the exit status of the previous command.

```
$ [ -t 1 ]
$ echo $?
0

$ [ -t 1 ] > hold
$ echo $?
1
```

As explained on page 302, the && (AND) control operator first executes the command preceding it. Only if that command returns a value of 0 (*true*) does && execute the command following it. In the following example, if **[–t 1]** returns 0, && executes **echo "FD 1 to screen"**. Although the parentheses (page 302) are not required in this example, they are needed in the next one.

```
$ ( [ -t 1 ] && echo "FD 1 to screen" )
FD 1 to screen
```

Next, the output from the same command line is sent through a pipeline to cat, so test returns 1 (*false*) and && does not execute echo.

```
$ ( [ -t 1 ] && echo "FD 1 to screen" ) | cat
$
```

The following example is the same as the previous one, except test checks whether file descriptor 2 is associated with the screen. Because the pipeline redirects only standard output, test returns 0 (*true*) and && executes echo.

```
$ ( [ -t 2 ] && echo "FD 2 to screen" ) | cat
FD 2 to screen
```

In this example, test checks whether file descriptor 2 is associated with the screen(it is) and echo sends its output to file descriptor 1 (which goes through the pipeline to cat).

Parameters

Shell parameters were introduced on page 310. This section goes into more detail about positional parameters and special parameters.

Positional Parameters

Positional parameters comprise the command name and command-line arguments. These parameters are called *positional* because you refer to them by their position on the command line. You cannot use an assignment statement to change the value of a positional parameter. However, the bash set builtin (page 472) enables you to change the value of any positional parameter except the name of the calling program (the command name). The tcsh set builtin does not change the values of positional parameters.

$0: Name of the Calling Program

The shell expands **$0** to the name of the calling program (the command you used to call the program—usually, the name of the program you are running). This parameter is numbered zero because it appears before the first argument on the command line:

```
$ cat abc
echo "This script was called by typing $0"
$ ./abc
This script was called by typing ./abc
$ /home/sam/abc
This script was called by typing /home/sam/abc
```

The preceding shell script uses echo to verify the way the script you are executing was called. You can use the basename utility and command substitution to extract the simple filename of the script:

```
$ cat abc2
echo "This script was called by typing $(basename $0)"
$ /home/sam/abc2
This script was called by typing abc2
```

When you call a script through a link, the shell expands **$0** to the value of the link. The busybox utility (page 747) takes advantage of this feature so it knows how it was called and which utility to run.

```
$ ln -s abc2 mylink
$ /home/sam/mylink
This script was called by typing mylink
```

When you display the value of **$0** from an interactive shell, the shell displays its name because that is the name of the calling program (the program you are running).

```
$ echo $0
bash
```

bash versus –bash

tip On some systems, **echo $0** displays **–bash** while on others it displays **bash**. The former indicates a login shell (page 288); the latter indicates a shell that is not a login shell. In a GUI environment, some terminal emulators launch login shells while others do not.

$1–$*n*: POSITIONAL PARAMETERS

The shell expands **$1** to the first argument on the command line, **$2** to the second argument, and so on up to **$*n***. These parameters are short for **${1}**, **${2}**, **${3}**, and so on. For values of ***n*** less than or equal to 9, the braces are optional. For values of ***n*** greater than 9, the number must be enclosed within braces. For example, the twelfth positional parameter is represented by **${12}**. The following script displays positional parameters that hold command-line arguments:

```
$ cat display_5args
echo First 5 arguments are $1 $2 $3 $4 $5

$ ./display_5args zach max helen
First 5 arguments are zach max helen
```

The **display_5args** script displays the first five command-line arguments. The shell expands each parameter that represents an argument that is not present on the command line to a null string. Thus, the **$4** and **$5** parameters have null values in this example.

Always quote positional parameters

caution You can "lose" positional parameters if you do not quote them. See the following text for an example.

Enclose references to positional parameters between double quotation marks. The quotation marks are particularly important when you are using positional parameters as arguments to commands. Without double quotation marks, a positional parameter that is not set or that has a null value disappears:

```
$ cat showargs
echo "$0 was called with $# arguments, the first is :$1:."

$ ./showargs a b c
./showargs was called with 3 arguments, the first is :a:.

$ echo $xx

$ ./showargs $xx a b c
./showargs was called with 3 arguments, the first is :a:.
$ ./showargs "$xx" a b c
./showargs was called with 4 arguments, the first is ::.
```

The **showargs** script displays the number of arguments it was called with (**$#**) followed by the value of the first argument between colons. In the preceding example, **showargs** is initially called with three arguments. Next, the echo command shows that the **$xx** variable, which is not set, has a null value. The **$xx** variable is the first argument to the second and third **showargs** commands; it is not quoted in the second command and quoted using double quotation marks in the third command. In the second **showargs** command, the shell expands the arguments to **a b c** and passes **showargs** three arguments. In the third **showargs** command, the shell expands the arguments to **"" a b c**, which results in calling **showargs** with four arguments. The difference in the two calls to **showargs** illustrates a subtle potential problem when using positional parameters that might not be set or that might have a null value.

set: Initializes Positional Parameters

When you call the set builtin with one or more arguments, it assigns the values of the arguments to the positional parameters, starting with **$1** (not in tcsh). The following script uses set to assign values to the positional parameters **$1**, **$2**, and **$3**:

```
$ cat set_it
set this is it
echo $3 $2 $1
$ ./set_it
it is this
```

optional A single hyphen (–) on a set command line marks the end of options and the start of values the shell assigns to positional parameters. A – also turns off the **xtrace** (**–x**) and **verbose** (**–v**) options (Table 8-13 on page 361). The following set command turns on **posix** mode and sets the first two positional parameters as shown by the echo command:

```
$ set -o posix - first.param second.param
$ echo $*
first.param second.param
```

A double hyphen (––) on a set command line without any following arguments unsets the positional parameters; when followed by arguments, **––** sets the positional parameters, including those that begin with a hyphen (–).

```
$ set --
$ echo $*

$
```

Combining command substitution (page 371) with the set builtin is a convenient way to alter standard output of a command to a form that can be easily manipulated in a shell script. The following script shows how to use date and set to provide the date in a useful format. The first command shows the output of date. Then cat displays the contents of the **dateset** script. The first command in this script uses command substitution to set the positional parameters to the output of the date utility. The next command, **echo $***, displays all positional parameters resulting from the previous set. Subsequent commands display the values of **$1**, **$2**, **$3**, and **$6**. The final command displays the date in a format you can use in a letter or report.

```
$ date
Tues Aug 15 17:35:29 PDT 2017
$ cat dateset
set $(date)
echo $*
echo
echo "Argument 1: $1"
echo "Argument 2: $2"
echo "Argument 3: $3"
echo "Argument 6: $6"
echo
echo "$2 $3, $6"

$ ./dateset
Tues Aug 15 17:35:34 PDT 2017

Argument 1: Tues
Argument 2: Aug
Argument 3: 15
Argument 6: 2017

Aug 15, 2017
```

You can also use the +*format* argument to date (page 787) to specify the content and format of its output.

set displays shell variables When called without arguments, set displays a list of the shell variables that are set, including user-created variables and keyword variables. Under bash, this list is the same as that displayed by declare (page 315) when it is called without any arguments.

```
$ set
BASH_VERSION='4.2.24(1)-release'
COLORS=/etc/DIR_COLORS
COLUMNS=89
LESSOPEN='||/usr/bin/lesspipe.sh %s'
LINES=53
LOGNAME=sam
MAIL=/var/spool/mail/sam
MAILCHECK=60
...
```

The bash set builtin can also perform other tasks. For more information refer to “set: Works with Shell Features, Positional Parameters, and Variables” on page 484.

shift: Promotes Positional Parameters

The shift builtin promotes each positional parameter. The first argument (which was represented by **$1**) is discarded. The second argument (which was represented by **$2**) becomes the first argument (now **$1**), the third argument becomes the second, and so on. Because no "unshift" command exists, you cannot bring back arguments that have been discarded. An optional argument to shift specifies the number of positions to shift (and the number of arguments to discard); the default is 1.

The following **demo_shift** script is called with three arguments. Double quotation marks around the arguments to echo preserve the spacing of the output but allow the shell to expand variables. The program displays the arguments and shifts them repeatedly until no arguments are left to shift.

```
$ cat demo_shift
echo "arg1= $1    arg2= $2    arg3= $3"
shift
echo "arg1= $1    arg2= $2    arg3= $3"
shift
echo "arg1= $1    arg2= $2    arg3= $3"
shift
echo "arg1= $1    arg2= $2    arg3= $3"
shift

$ ./demo_shift alice helen zach
arg1= alice    arg2= helen    arg3= zach
arg1= helen    arg2= zach     arg3=
arg1= zach     arg2=    arg3=
arg1=     arg2=    arg3=
```

Repeatedly using shift is a convenient way to loop over all command-line arguments in shell scripts that expect an arbitrary number of arguments. See page 436 for a shell script that uses shift.

$* and $@: Expand to All Positional Parameters

The shell expands the **$*** parameter to all positional parameters, as the **display_all** program demonstrates:

```
$ cat display_all
echo All arguments are $*

$ ./display_all a b c d e f g h i j k l m n o p
All arguments are a b c d e f g h i j k l m n o p
```

"$*" versus "$@"

The **$*** and **$@** parameters work the same way except when they are enclosed within double quotation marks. Using **"$*"** yields a single argument with the first character in **IFS** (page 321; normally a SPACE) between the positional parameters. Using **"$@"** produces a list wherein each positional parameter is a separate argument. This difference typically makes **"$@"** more useful than **"$*"** in shell scripts.

The following scripts help explain the difference between these two parameters. In the second line of both scripts, the single quotation marks keep the shell from interpreting the enclosed special characters, allowing the shell to pass them to echo so echo can display them. The **bb1** script shows that **set "$*"** assigns multiple arguments to the first command-line parameter.

```
$ cat bb1
set "$*"
echo $# parameters with '"$*"'
echo 1: $1
echo 2: $2
echo 3: $3

$ ./bb1 a b c
1 parameters with "$*"
1: a b c
2:
3:
```

The **bb2** script shows that **set "$@"** assigns each argument to a different command-line parameter.

```
$ cat bb2
set "$@"
echo $# parameters with '"$@"'
echo 1: $1
echo 2: $2
echo 3: $3

$ ./bb2 a b c
3 parameters with "$@"
1: a
2: b
3: c
```

Special Parameters

Special parameters enable you to access useful values pertaining to positional parameters and the execution of shell commands. As with positional parameters, the shell expands a special parameter when it is preceded by a **$**. Also as with positional parameters, you cannot modify the value of a special parameter using an assignment statement.

$#: Number of Positional Parameters

The shell expands **$#** to the decimal number of arguments on the command line (positional parameters), not counting the name of the calling program:

```
$ cat num_args
echo "This script was called with $# arguments."
```

```
$ ./num_args sam max zach
This script was called with 3 arguments.
```

The next example shows set initializing four positional parameters and echo displaying the number of parameters set initialized:

```
$ set a b c d; echo $#
4
```

$$: PID Number

The shell expands the **$$** parameter to the PID number of the process that is executing it. In the following interaction, echo displays the value of this parameter and the ps utility confirms its value. Both commands show the shell has a PID number of 5209:

```
$ echo $$
5209
$ ps
  PID TTY          TIME CMD
 5209 pts/1    00:00:00 bash
 6015 pts/1    00:00:00 ps
```

Because echo is built into the shell, the shell does not create another process when you give an echo command. However, the results are the same whether echo is a builtin or not, because the shell expands **$$** *before* it forks a new process to run a command. Try giving this command using the echo utility (**/bin/echo**), which is run by another process, and see what happens.

Naming temporary files

In the following example, the shell substitutes the value of **$$** and passes that value to cp as a prefix for a filename:

```
$ echo $$
8232
$ cp memo $$.memo
$ ls
8232.memo memo
```

Incorporating a PID number in a filename is useful for creating unique filenames when the meanings of the names do not matter; this technique is often used in shell scripts for creating names of temporary files. When two people are running the same shell script, having unique filenames keeps the users from inadvertently sharing the same temporary file.

The following example demonstrates that the shell creates a new shell process when it runs a shell script. The **id2** script displays the PID number of the process running it (not the process that called it; the substitution for **$$** is performed by the shell that is forked to run **id2**):

```
$ cat id2
echo "$0 PID= $$"
$ echo $$
8232
$ ./id2
```

```
./id2 PID= 8362
$ echo $$
8232
```

The first echo displays the PID number of the interactive shell. Then **id2** displays its name (**$0**) and the PID number of the subshell it is running in. The last echo shows that the PID number of the interactive shell has not changed.

$!: PID Number of Most Recent Background Process

The shell expands **$!** to the value of the PID number of the most recent process that ran in the background (not in tcsh). The following example executes sleep as a background task and uses echo to display the value of **$!**:

```
$ sleep 60 &
[1] 8376
$ echo $!
8376
```

$?: Exit Status

When a process stops executing for any reason, it returns an *exit status* to its parent process. The exit status is also referred to as a *condition code* or a *return code*. The shell expands the **$?** (**$status** under tcsh) parameter to the exit status of the most recently executed command.

By convention, a nonzero exit status is interpreted as *false* and means the command failed; a zero is interpreted as *true* and indicates the command executed successfully. In the following example, the first ls command succeeds and the second fails; the exit status displayed by echo reflects these outcomes:

```
$ ls es
es
$ echo $?
0
$ ls xxx
ls: xxx: No such file or directory
$ echo $?
1
```

You can specify the exit status a shell script returns by using the exit builtin, followed by a number, to terminate the script. If you do not use exit with a number to terminate a script, the exit status of the script is that of the last command the script ran.

```
$ cat es
echo This program returns an exit status of 7.
exit 7
$ es
This program returns an exit status of 7.
$ echo $?
7
```

```
$ echo $?
0
```

The **es** shell script displays a message and terminates execution with an exit command that returns an exit status of 7, the user-defined exit status in this script. The first echo then displays the exit status of **es**. The second echo displays the exit status of the first echo: This value is 0, indicating the first echo executed successfully.

$–: Flags of Options That Are Set

The shell expands the **$–** parameter to a string of one-character bash option flags (not in tcsh). These flags are set by the set or shopt builtins, when bash is invoked, or by bash itself (e.g., **–i**). For more information refer to "Controlling bash: Features and Options" on page 359. The following command displays typical bash option flags for an interactive shell:

```
$ echo $-
himBH
```

Table 8-13 on page 361 lists each of these flags (except **i**) as options to set in the **Alternative syntax** column. When you start an interactive shell, bash sets the **i** (interactive) option flag. You can use this flag to determine if a shell is being run interactively. In the following example, **display_flags** displays the bash option flags. When run as a script in a subshell, it shows the **i** option flag is not set; when run using source (page 290), which runs a script in the current shell, it shows the **i** option flag is set.

```
$ cat display_flags
echo $-

$ ./display_flags
hB

$ source ./display_flags
himBH
```

$_: Last Argument of Previously Executed Command

When bash starts, as when you run a shell script, it expands the **$_** parameter to the pathname of the file it is running. After running a command, it expands this parameter to the last argument of the previously executed command.

```
$ cat last_arg
echo $_
echo here I am
echo $_

$ ./last_arg
./last_arg
```

```
here I am
am
```

In the next example, the shell never executes the echo command; it expands $_ to the last argument of the ls command (which it executed, albeit unsuccessfully).

```
$ ls xx && echo hi
ls: cannot access xx: No such file or directory
$ echo $_
xx
```

The tcsh shell expands the $_ parameter to the most recently executed command line.

```
tcsh $ who am i
sam      pts/1        2018-02-28 16:48 (172.16.192.1)
tcsh $ echo $_
who am i
```

VARIABLES

Variables, introduced on page 310, are shell parameters denoted by a name. Variables can have zero or more attributes (page 315; e.g., export, readonly). You, or a shell program, can create and delete variables, and can assign values and attributes to variables. This section adds to the previous coverage with a discussion of the shell variables, environment variables, inheritance, expanding null and unset variables, array variables, and variables in functions.

SHELL VARIABLES

By default, when you create a variable it is available only in the shell you created it in; it is not available in subshells. This type of variable is called a *shell variable*. In the following example, the first command displays the PID number of the interactive shell the user is working in (2802) and the second command initializes the variable **x** to **5**. Then a bash command spawns a new shell (PID 29572). This new shell is a child of the shell the user was working in (a subprocess; page 333). The **ps –l** command shows the PID and PPID (parent PID) numbers of each shell: PID 29572 is a child of PID 2802. The final echo command shows the variable **x** is not set in the spawned (child) shell: It is a shell variable and is local to the shell it was created in.

```
$ echo $$
2802
$ x=5
$ echo $x
5
```

```
$ bash
$ echo $$
29572
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2802  2786  0  80   0 -  5374 wait   pts/2    00:00:00 bash
0 S  1000 29572  2802  0  80   0 -  5373 wait   pts/2    00:00:00 bash
0 R  1000 29648 29572  0  80   0 -  1707 -      pts/2    00:00:00 ps
$ echo $x

$
```

Environment, Environment Variables, and Inheritance

This section explains the concepts of the command execution environment and inheritance.

Environment

When the Linux kernel invokes a program, the kernel passes to the program a list comprising an array of strings. This list, called the *command execution environment* or simply the *environment,* holds a series of name-value pairs in the form *name=value.*

Environment Variables

When bash is invoked, it scans its environment and creates parameters for each name-value pair, assigning the corresponding *value* to each *name.* Each of these parameters is an *environment variable;* these variables are in the shell's environment. Environment variables are sometimes referred to as *global variables* or *exported variables.*

Inheritance A child process (a subprocess; see page 333 for more information about the process structure) inherits its environment from its parent. An inherited variable is an environment variable for the child, so its children also inherit the variable: All children and grandchildren, to any level, inherit environment variables from their ancestor. A process can create, remove, and change the value of environment variables, so a child process might not inherit the same environment its parent inherited.

Because of process locality (next), a parent cannot see changes a child makes to an environment variable and a child cannot see changes a parent makes to an environment variable once the child has been spawned (created). Nor can unrelated processes see changes to variables that have the same name in each process, such as commonly inherited environment variables (e.g., **PATH**).

Process Locality: Shell Variables

Variables are *local,* which means they are specific to a process: *Local* means *local to a process.* For example, when you log in on a terminal or open a terminal emulator, you start a process that runs a shell. Assume in that shell the **LANG** environment variable (page 327) is set to **en_US.UTF-8.**

If you then log in on a different terminal or open a second terminal emulator, you start another process that runs a different shell. Assume in that shell the **LANG** environment variable is also set to **en_US.UTF-8**. When you change the value of **LANG** on the second terminal to **de_DE.UTF-8**, the value of **LANG** on the first terminal does not change. It does not change because variables (both names and values) are local to a process and each terminal is running a separate process (even though both processes are running shells).

export: PUTS VARIABLES IN THE ENVIRONMENT

When you run an **export** command with variable names as arguments, the shell places the names (and values, if present) of those variables in the environment. Without arguments, export lists environment (exported) variables.

Under tcsh, setenv (page 396) assigns a value to a variable and places the name (and value) of that variable in the environment. The examples in this section use the bash syntax but the theory applies to both shells.

The following **extest1** shell script assigns the value of **american** to the variable named **cheese** and then displays its name (the shell expands **$0** to the name of the calling program) and the value of **cheese**. The **extest1** script then calls **subtest**, which attempts to display the same information, declares a **cheese** variable by initializing it, displays the value of the variable, and returns control to the parent process, which is executing **extest1**. Finally, **extest1** again displays the value of the original **cheese** variable.

```
$ cat extest1
cheese=american
echo "$0 1: $cheese"
./subtest
echo "$0 2: $cheese"

$ cat subtest
echo "$0 1: $cheese"
cheese=swiss
echo "$0 2: $cheese"

$ ./extest1
./extest1 1: american
./subtest 1:
./subtest 2: swiss
./extest1 2: american
```

The **subtest** script never receives the value of **cheese** from **extest1** (and **extest1** never loses the value): **cheese** is a shell variable, not an environment variable (it is not in the environment of the parent process and therefore is not available in the child process). When a process attempts to display the value of a variable that has not been declared and is not in the environment, as is the case with **subtest**, the process displays nothing; the value of an undeclared variable is that of the null string. The final echo shows the value of **cheese** in **extest1** has not changed: In bash—unlike in the real world—a child can never affect its parent's attributes.