Shell scripts in 20 pages

A guide to writing shell scripts for C/C++/Java and unix programmers

Russell Quong

Jan 5 2005 - Document version 2002a

Keywords: Shell documentation, Shell tutorial, Shell beginners, Guide to shell scripts. (For internet search engines.)

This is a work in progess; you will find unfinished sections, paragraphs and even sentences.

Table of Contents

This is a work in progess; you will find unfinished sections, paragraphs and even sentences.

Introduction

This document assumes you are using bash version 2; most of the examples will work for sh and ksh too.

A PDF version of this file is at http://www.quong.com/shellin20/shellin20.pdf .

What is a shell?

A shell is a program that reads commands and executes them. Another name for a shell is a *command interpreter*. For those who only know Windows: MS-DOS is a shell.

Current Unix shells, such as bash, ksh, and tcsh, provide numerous conveniences and features for both interactive and programmatic purposes and are complicated programs in their own right. The `bash` manual page is over 50 pages of dense documentation. Finally, if all you have used is MS-DOS, be aware it is an extremely primitive shell.

There are many (30+) Unix shells, but the most popular are `sh`, `ksh`, `bash`, `csh` and `tcsh`.

My own history with Unix shells

I started using `csh` many years ago as an undergraduate, because I was too stupid to figure out the `/bin/sh` syntax, in particular `${var:-val}`. Despite encountering many mysterious `/bin/sh` scripts and having to use `make`, which uses `/bin/sh`, I resisted `sh` and wrote csh shell scripts and used the `tcsh` as my login shell. Finally, in 1999, I couldn't stand `csh` scripting any more, and "re"-learned /bin/sh.

Useful links

Short overview of different shells	http://www.faqs.org/faqs/unix-faq/shell/shell-differences/
Short overview of different shells	http://www.faqs.org/faqs/unix-faq/faq/part5/
List of csh problems	http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/

The Operating System

A typical home user is completetely insulated from the OS. So when someone says, "I really like computer XXX (e.g. the Mac or Windows 95 or Unix)", they are NOT talking about the operating system. Rather they are talking about the user interface or *UI* on top of the OS.

Externally, an operating system is a programming API, typically in the C programming language. The API lets some other program do low level operations like:

Unix API call	Description
exec	run a program, given a fully specified command
open	open file or some other I/O stream
read/write	read or write data to a file descriptor

Directly interacting with the OS is incessantly tedious, as the OS is very picky and works at a low level. Its akin to communication via Morse Code.

Instead, people use graphical environments (like the Mac or Win32) or command line interpreters like Unix *shells* or the (very minimal) MSDOS prompt.

Why program in the shell instead of (Perl, Python, etc)?

1) You may not have Perl or Python available. E.g. for system administration, when the system is first coming up, the shell maybe your only option.

2) For simple file based manipulation, the shell is terser and cleaner than Perl.

Interactive versus scripts

When you manually type commands to the shell, you are running an *interactive shell*. The shell features beneficial for interactive use are different from those needed when running a script, in which the shell reads commands from a file.

In interactive use, shell features that minimize typing and tedium are important. For scripting or programmatic use, flexibility, power and expressiveness are more imporant.

Overall evaluation

Shell	Interactive	Scripting
sh	C-	B
ksh	B+	A-
bash	A	A
csh	B+	C-
tcsh	A	C+
zsh	A- (?)	A (?)
rc/es	A- (?)	A (?)

Command processing I

Consider how the shell processes the following command.

```
% ls -l *.c > listing.out
```

1. Split the command into words based on whitespace. Here, there are three words (ls) (-l) and (*.c) before the redirection (>) word. Each word is processed separately.
2. Set aside the redirection specification (> listing.out).
3. We redirect standard out to the file `listing.out`.
4. We apply globbing (or pathname expand) on the `*` in the `*.c` word, replacing *.c with matching file names, say `apple.c`, `banana.c` and `cherry.c`. The command now consists of the words (ls) (-l) (apple.c) (banana.c) (cherry.c).
5. The first word (ls) is the program to run. We search each directory in the `PATH` variable for an executable `ls` file; we execute the first one we find.

We can break up the command into parts as follows.

Term	What	Example
program	first word in command	ls
flags	options that affect the command	-l
arguments	all words but the program	-l *.c

Interactive Use of shells

For interactive use, I prefer `bash` and `tcsh`, because they have easily accessible filename and command completion (via TAB) and good editing capabilities. Note the phrase *completes* means that if you partially type a word, the shell will either

On a ...	the shell does ...
unique match	finishes the rest of the word
multiple matches	shows all possible completions of the partial word

The features I rely on from most important to least important are

What	Keys	Description
filename completion	TAB	completes partially typted file,path names
command history access	CNTL-p (CNTL-n)	fetch the previous (next) command to edit or execute
command history search	CNTL-r phrase	(bash) reverse search for phrase (as you type it) through the history. CNTL-r again to skip back to the previous command matching. Just try it.
command completion	TAB	completes the command name
CDPATH		variable of directories to search when you type `cd`

ksh: To enable command editing in `ksh`, use `set -o emacs` or `set -o vi`. Skip the `fc`

Setting your prompt

Set the `PS1` (prompt string 1) variable. In `PS1`, the following escape sequences can be used. I have listed only the most useful; see the bash man page for a full listing.

\h	hostname	\u	user name
\H	hostname.domainname	\w	current working directory (CWD)
\n	newline	\W	basename of CWD
\r	carriage return	\!	history number of the current command
\s	shell name	$	if UID is 0 (root), use a '#', else use a '$'

I personally set

```
  PS1='\h \! \w\\$ '
  crank 647 ~/src/template$ ls            # my prompt before 'ls'
```

Real or physical paths (bash)

In the presense of symbolic links and home directories, bash by default uses the logical directory structure. To force bash to show the actual, real or physical directory structure use `cd -P <dir>`; I alias `cdp` to `cd -P .` . As an example if, `/home/quong` is a symbolic link to `/box22/disk88/c/quong`, then

```
  % cd /home/quong      # current dir as /home/quong
  % cd ..               # cd to /home
  % cd -P /home/quong   # current dir as /box22/disk88/c/quong
  % cd ..               # cd to /box22/disk88/c
```

Processes or jobs or tasks

Each command run by a shell is a separate child process, whether run interactively or via a script. The child process inherits various values, such as *(i)* who is running the command, *(ii)* the current directory, and *(iii)* the environment variables.

Variables

Shell variables contain string values, though you can force the values to be used numerically. Normal variables are local/private to one shell job/process and are only accessible (or visible) to the shell in which they are set. If you a globally visible variable, you must `export` it.

Assign to variables using `=` *with no surrounding space* between variable name and value. Access the value of variable by using a $ before the variable name, e.g. `$showWarnings`.

```
 % color=red                     # correct
 % color= red                    # WRONG, space after equal
 % echo I want a ${color}der than $color shirt  # I want a redder than red  shirt
```

If there is any ambiguity what the variable name is, you can use `${varname}`. In the preceding example, see how we echo'ed "redder".

Because the shell uses space to break up commands, to store a string value with a space in a variable, use quotes.

```
  % colors="red green blue"
  % for c in $colors ; do echo $c ; done
red
green
blue
  % for c in "$colors" ; do echo $c ; done
red green blue
```

Environment (or public) variables

Public or *environment variables* are accessible by all child processes/jobs of the shell.

Common environment variables

PATH	dirs to search for commands
SHELL	path of shell
TERM	terminal type
USER	user (login) name
HOME	home dir of user
PS1	main interactive prompt (ba/k/sh)
CDPATH	dirs to search when you do a cd or pushd

Scripting

(To be done.)

True and false

For various control constructs, like `if`, `while`, `&&` and `||`, the shell runs a command and the command has an exits or returns either true (success) and false (failure). Every command in Unix has an exit value. However, unlike C/C++, true is 0 (zero) and false is anything else, non-zero. I remember is this notation because there is only one way for a command to succeed but there are many ways a command can fail (no such file, missing permissions, out of disk space, bad name, and many others).

Each command returns a (normally) hidden integer value. In C/C++ programs, the return value of `int main()` or the parameter passed to the `exit()` function. Here is the C source code for a that always returns true.

```
int main (int argc, char* [] argv) { return 0; }
```

The special variable `$?` contains the exit status of the last command run, however you should rarely have to access this variable. Evaluate the command in the if/while directly. The following example shows how to process a file `$fx` if it contains a java class.

```
  # poor, too wordy
grep -c ~class $fx > /dev/null
if [ $? = 0 ]; then
  process $fx
fi


  # much better, directly run grep
```

```
if grep -c ^class $fx > /dev/null; then
  process $fx
fi
```

Relational operators

If

The `if` construct looks as follows with an optional else and multiple optional elif (else if).

```
if EXPR; then

   body

fi

if [ -d $ff ]; then

   echo "Dir: $ff"

fi


if [ -d $ff ]; then

   echo "Directory: $ff, here is the total size:"

   du -s $ff

elif [ -f $ff ]; then

   echo "File: $ff"

else

   echo "What the heck is $ff?"

   ls -l $ff

fi
```

While

The `while` command is the only way to loop in the shell. The looping continues so long as the EXPR returns true.

```
  while EXPR ; do
    body
  done
```

For example, to store all the command line arguments but the last one in `allbutlast` in a script

```
#! /bin/bash

allbutlast=""
while [ $# -gt 1 ]; do
  allbutlast="$allbutlast $1"
  shift
done
last=$1
shift
```

For

The `for` construct is a "foreach" in that a variable is assigned each value in a list one by one. For example, to find out which files in the subdirectory `infodir` are text files, we run the `file` command and grep for the word `text`.

```
  for VAR in LIST ; do
    body
  done

  for ff in infodir/* ; do
    if file $ff | grep text > /dev/null ; then
      echo "File $ff is text"
    fi
  done
```

Case

The case statement lets you determine if a string SSS, which is almost always contained by a variable VVV, matches any of several "cases". For example we test which state a traffic signal is in via:

```
  case $trafficLight in
    red ) echo "stop" ;;
    yellow | orange ) echo "decision time..." ;;
    green ) echo "GO" ;;
    default ) echo "Unknown color ($trafficLight)" ;;
  esac
```

The `case` construct is the only way to apply glob matching to arbitrary strings. The following example ask the user a Yes-no question and then treats any response beginning with a 'y' or 'Y' as a "yes". Also, any response starting with a 'q' quits out. Also, note that the `break` statement breaks out of the `while` loop, not the `case`, unlike C/C++/Java.

```
while true; do
  echo -n "list the current dir? (y/n) "
  read yn
  case $yn in
    y* | Y* ) ls -l . ; break ;;
    [nN]* )   echo "skipping" ; break ;;
    q* ) exit ;;
    * ) echo "unknown response.  Asking again" ;;
  esac
done
```

Syntax of control structures

It is possible to write any shell script in a single line. In practice, it is sometimes convenient to do so. For example, in a makefile, shell commands spanning more than one line are ugly and error prone.

When processing control constructions, the ba/k/sh shells need a *delimiter*, either a `newline` or a `;` (semi-colon), to terminate arbitrary commands. Thus, after the keywords `if, then, do, while` we do not need a delimiter, but before the `fi` or `done`, we need a delimeter indicating the end of the previous command. As an example, remove the delimiters from the following legal (!) command to see the ensuing confusion.

```
  % if echo then fi if then ; then ls fi fi ; fi
```

Thus in the following, DELIM means either a `newline` or a `;` delimiter. Thus the following four if-statements are all equivalent.

```
  if EXPR DELIM then STMT(S) DELIM fi   # general syntax
  if [ -f /bin/mv ]DELIM then echo "looks like unix" DELIM fi
  if [ -f /bin/mv ]; then echo "looks like unix" ; fi
  if [ -f /bin/mv ]; then
    echo "looks like unix"
  fi
  if [ -f /bin/mv ]
      then
    echo "looks like unix" ; fi
```

The syntax for control constructs is

if	if EXPR DELIM then STMT(S) DELIM fi
if else	if EXPR DELIM then STMT(S) DELIM elif EXPR ; then STMT(S) DELIM fi
for	for VAR in LIST DELIM do STMT(S) DELIM done
while	while EXPR DELIM do STMT(S) DELIM done
case	case VALUE in [[PATTERN [| PATTERNS]) STMTS ;;] esac

Testing if it is an interactive shell

All shells read a startup file, in which you can set and customize various settings (variables, aliases, functions, prompt, terminal settings). There are three cases to consider, (a) when shell handles an interactive login, (b) when a remote shell runs a command and (c) when the shell reads a script.

When you interactively "login" you get an interacitve shell and you probably want to (heavily) customize its use. However if you run a command remotely, say via the Unix rsh, rcp or rsync commands, you start a non-interactive remote shell to run the remote command and you usually to set the path correctly. In particular, you **must not** print any messages when the remote shell start up.

To test if a shell is interactive, *(i)* test for the existance of the shell prompt string variable `PS1` or *(ii)* run `tty -s` which returns true (0) for an interactive shell, as there is an underlying tty.

Conditional Tests

To perform a conditional test on files, strings or numbers, use either `[ expr ]` or `test expr` as in the following two examples.

```
if [ -f file.txt ]; then ... ; fi
if test -f file; then ... ; fi
```

The following table shows the conditional tests provided by bash from most to least common in this authors experience. Some descriptions are directly from the bash man page.

String operations	
string1 = string2	True if the strings are equal.
string1 != string2	True if the strings are not equal.
-z string	True if the length of string is zero.
string	True if the length of string is non-zero.
-n string	True if the length of string is non-zero.
string1 == string2	(Bash only) True if the strings are equal.
-o optname	True if shell option optname is enabled. See the list of options under the description of the -o option to the set builtin below.

Numeric operations	
arg1 OP arg2	OP is one of -eq, -ne, -lt, -le, -gt, or -ge. These arithmetic binary operators return true if arg1 is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to arg2, respectively. Arg1 and arg2 may be positive or negative integers.
string1 < string2	True if string1 sorts before string2 lexicographi- cally in the current locale.
string1 > string2	True if string1 sorts after string2 lexicographi- cally in the current locale.

File operations	
-e file	True if file exists.
-d file	True if file exists and is a directory.
-f file	True if file exists and is a regular file.

-L file	True if file exists and is a symbolic link.
-r file	True if file exists and is readable.
-w file	True if file exists and is writable.
-x file	True if file exists and is executable.
file1 -nt file2	True if file1 is newer (according to modification date) than file2.
file1 -ot file2	True if file1 is older than file2.
file1 -ef file2	True if file1 and file2 have the same device and inode numbers.
Less frequently used operations	
-a file	True if file exists.
-b file	True if file exists and is a block special file.
-c file	True if file exists and is a character special file.
-g file	True if file exists and is set-group-id.
-h file	True if file exists and is a symbolic link.
-k file	True if file exists and its "sticky" bit is set.
-p file	True if file exists and is a named pipe (FIFO).
-s file	True if file exists and has a size greater than zero.
-t fd	True if file descriptor fd is open and refers to a terminal.
-u file	True if file exists and its set-user-id bit is set.
-O file	True if file exists and is owned by the effective user id.
-G file	True if file exists and is owned by the effective group id.
-S file	True if file exists and is a socket.
-N file	True if file exists and has been modified since it was last read.

Clarification A

Both the `if` and the `while` constrol constructs take commands. However, what about the common syntax `if [ expr ]; ...`? The simple but non-obvious answer is that `[` (yes, left bracket) is a (built-in) command, which parses its arguments. The right bracket argument is needed for the command

Conditional Assignment

Many times we want a conditionally assign a value to a variable VVV. The syntax `VVV=${ZZZ:-DefaultVal}` is equivalent to

```
VVV=${ZZZ:-DefaultVal\}
     # same as
if [ "$ZZZ" != "" ]; then
  VVV=$ZZZ
else
  VVV=DefaultVal
fi
```

Thus we assign the value of `$ZZZ` to VVV if `ZZZ` has a value, otherwise we assign DefaultVal.

I/O redirection

One strength of Unix and its shells is the ability to redirect I/O to/from files and or other commands. For example, to see the 5 newest files in the directory DDD, we list the files sorted by time (ls -t) and select the first 6 lines (head -6) via:

```
  ls -t DDD | head -6
```

Deep down in Unix, all files are refereced by a integer *file descriptor*, which is the index into a table of the open streams (files) that each process has. There are three standard pre-opened streams in Unix (actually, the shell pre-opens these three streams.)

File Desc	name	by default
0	stdin	keyboard
1	stdout	screen, buffered, not-flushed
2	stderr	screen, always flushed

The I/O redirection directives are:

> filename	send stdout to the file `filename`
n> filename	redirect FD `n` to the file `filename`
n>&k	redirect FD `n` to FD `k`
/ command	send stdout (FD 1) to the program `command`

The shell processes directives in order from left to right. This is significant for cases where you want to redirect both stdout and stderr. We explain via the examples below. And while some of the examples may seem contrived, this author has used all the examples trying to get real work done.

ls > /tmp/list	send ls output to `/tmp/list`
ls > /tmp/list 2> ./err	as above, but send stderr to `./err`
ls > /tmp/list 2>&1	send both stdout and stderr to `/tmp/list`
ls 2>&1 > /tmp/list	send stdout to `/tmp/list` and stderr to the screen via the default stdout stream
ls 2>&1 | less	send both stdout and stderr to `less`

Here is a shell function `echoerr` that echos its arguments to `stderr` instead of `stdout`. It is useful for generating error messages in a large script.

```
 \% echoerr () { echo "$@" 1>&2 ; }
 \% echoerr "Oooh.  Not good."
```

Debugging Scripts

1. Use `echo` statements.
2. Run `bash -n script` to check for syntax errors.
3. Use the command `set -v` to get a verbose dump of each line the shell reads. Use `set +v` to turn off verbose mode.
4. Use the command `set -x` to see what each command expands to. Again, `set +x` turns this mode off.

Command line argument processing

The command line parameters to a script are stored in the nearly identical variables `$*` and `$`. The following table summarizes the variables you would use for command line processing. For the example values, assume you wrote a ba/k/sh script `/usr/bin/args.sh` and ran it as shown below.

Variable	Meaning	Ex: echoArgs -t two "let's go"
`$*`	Command line args	-t two let's go
`$@`	Command line args	-t two "let's go"
`$#`	Number of args	3
`$0`	Name of script	/usr/bin/args.sh
`$1`	First arg in $*	-t
`$2`	Second arg in $*	two
`$3`	Third arg in $*	let's go
`$4`	Fourth arg in $*	(empty)

The following shell function echoArgs shows the difference between `$*` and `$@`. To use `$@` in a for loop, you must put it in double quotes.

```
echoArgs () {
    echo $#
    for i in "$@"; do
        echo "($i)";
    done;
    for i in $*; do
        echo "(($i))";
    done
}
$ echoArgs -t two "let's go"
  3
  (-t)
  (two)
  (let's go)
  ((-t))
  ((two))
  ((let's))
  ((go))
```

To parse command line arguments, I prefer using the `case` construct, as shown below, instead of the builtin `getopts` in `ba/k sh`, because case is easier to understand, handles all flag situations, and will work in `sh` too.

Here is a more realistic example for a command that takes five possible flags, in any order. For the `-n` flag, we set a shell variables to remember the state; this technique is common.

Flag	Description
-o OUT	send output to file OUT
-n	show what you would do but do not do it
-v	give more output, each -v increases verboseness
-l	same as -verbose
-version	show the version and quit

Here is the code snippet. Notice the `shift 2` and the `$2` for the -o flag. Notice that any flag beginning `-ver` is considered the same are `-version`.

```
nflag=0
vlevel=0
OUT=
while [ $# -gt 0 ]; do
  case "$1" in
   -o ) OUT=$2 ; shift 2 ;;
   -n ) nflag=1 ; shift ;;
   -l | -v ) vlevel=$(( vlevel+1 )) ; shift ;;
   -ver* ) echo "Version $version"  ; exit 1 ;;
   * ) echo "Saw non flag $arg" ; break ;;
  esac
done
... continue processing remaining args ...
```

Special variables

$	shell's process ID, e.g. tempfile=/tmp/out.$$
PATH	command search path
CDPATH	cd search path
...	more to be added

Tilde, brace, and globbing expansions

The shells will expand the following strings

Expansion	You type	the shell generates
Tilde	~	your home directory (`$HOME`)
Tilde	~alison	home directory for user `alison`
Brace	{1,blue,dot.com}	1 blue dot.com
Brace	x{0,11}y{2,33,}z	x0y2z x0y33z x0yz x11y2z x11y33z x11yz

Globbing and Filename expansion

On the command line, the shell does filename expansion replacing `*.pdf` will all filenames ending in `.pdf`. There are two separate concepts being used. The first, called glob matching or *globbing*, means that some characters like `*` have special meaning. The second concept is that globbing is being applied to filenames. Because the two are used almost synomously, most people think incorrectly think globbing only applies to file names. However, the case statement uses globbing on an arbitrary string.

On a command line, the following characters have special meaning. This process is called *globbing*.

*	Any sequence of characters not containing a `/`
?	Any single character
[aeiou]	Any single a, e, i, o or u character
[^aeiou]	Any character *except* a, e, i, o or u

A leading `*` or `?` will not match a leading dot (`.`) to prevent from * from matching `.` and `..` which would normally cause havoc. To match files like `.profile`, you can use the glob pattern `.*`.

Filename expansion

The shell applies does applies globbing to all command line arguments matched against filenames starting in the current directory. Thus */*.pdf matches all `.pdf` files in all subdirectories of the current directory.

Arithmetic

In ksh,bash use `$(( expression ))` to perform arithmetic operations. Note that inside `$(( expression ))`, you do not need to prefix variables with a $.

```
echo 'using $(( var + 1 )) style'
i=0 j=0 k=0 ll=0
while [ $i -le 4 ]; do
  echo $i $j $k $ll
  i=$(( $i + 1 ))          # OK to use $i
  j=$(( j + i ))           # just 'j' is fine, too
  ll=$(( k += i ))
```

```
done
```

Back tick expansion or command substitution

The notation `` `command` `` (we use back quotes not the normal forward quotes) or `$(command)` is replaced by the output of the command. Typically command only produces one line of output. For example the `basename PATH` command strips off any directory portion of path, so to get the file name in a script you usually see:

```
  fff=/usr/share/timezone/Pacific.tz
  filepart=`basename $fff`      # filepart=Pacific.tz
```

To set a variable value to be the contents of a file, you can use either of

```
  hostname=`cat /etc/HOSTNAME`
  hostname=$(< /etc/HOSTNAME)         # special form
```

Embedding verbatim text with here documents

If you need to print out text nearly verbatim, e.g. you need to generate a standard 40-line disclaimer, then use a *here document*. The general notation is as follows, where you can use any string of your choice to replace `END_DELIMITER`.

```
  some-previous-shell-command
  cat <<-END_DELIMITER
    verbatim text
  ...
END_DELIMITER

  cat <<-'END_DELIMITER'             # single quote variation
  cat <<-"END_DELIMITER"             # double quote variation
```

The optional minus sign - before END_DELIMITER tells bash to ignore beginning *tabs* in each line of the verbatim text, so you can indent this text. The shell will evaluate shell variables, backtick expansion and (bash) arithmetic expressions in the verbatim text. To suppress this evaluation, put single or double quotes around END_DELIMITER, as shown and each line of the here document will be treated as if it had been quoted that way.

Here is a more realistic example, where we generate an HTML header to the file `$out`.

```
shell_function() {
  htmldoc=...
  cat >> $out <<-EOS
        <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
        <HTML><HEAD>
        <TITLE>The HMTL doc: $htmldoc on `date`</TITLE>
        </HEAD>
EOS
```

```
  ...
}
```

Process management

The shell can manage several processes (jobs/tasks). The current or *selected* job (which must be suspended or running the background) is marked with a `+` when you type `jobs`. A foreground process is one that is currently running and has control of the terminal. E.g. your keyboard input goes to the foreground process.

Command	What
jobs	list the jobs running on this shell
bg [Proc]	run selected job in the background
fg [Proc]	run selected job in the foreground
CTL-Z	suspend the current job

For example, I had three netscape's and some other processes running and typed jobs and got:

```
hostest 972 ~/bin$ netscape &      # this was the fifth job/task
hostest 973 ~/bin$ fg 6            # resume running less
CTL-Z                              # suspend less
hostest 974 ~/bin$ jobs
[1]   Running                 ( cd ~; netscape-v304 ) &  (wd: ~)
[3]   Running                 oclock &  (wd: ~)
[4]   Running                 netscape -geometry =720x700 &  (wd: ~/ftp)
[5]-  Running                 netscape -geometry =720x700 &  (wd: ~/ftp)
[6]+  Stopped                 less -c -s -M ../summary-01-21-01.out  (wd: /tmp/tmp/tmp)
hostest 974 ~/bin$ fg
```

In a shell script, the `wait` command will wait for all background jobs to finish before proceeding.

```
  ...
  commandOne -a -x -b &
  commandTwo -vv file1 file2 &
  wait    # waits for the two previous commands to finish
  ...
```

Useful commands

When writing scripts, I found the following commands particularly useful. See the respective man page for all the options.

uname	get system info
basename	strip off dir component
dirname	strip off file component

date	get date in arbitrary format
sed	stream editor; use for regex support
...	more to be added

Reading input

Some useful functions

Tips for writing scripts

To discard output, send it to `/dev/null` in Unix.

Tips, tricks and examples

To process a listing of files `*` here are three way from worst to best.

```
  # yech
filelist=`ls *`
for ff in $filelist; do ... ; done


  # poor
filelist=`echo *`
for ff in $filelist; do ... ; done


  # best
for ff in *; do ... ; done
```

Seeing variables

Here is a handy ba/k/sh function that prints out the values of variables given their *names*. I list it first since I use it often.

```
   # showVals varname [ varname(s) ]
showVals () {
  for i in $*; do
    eval echo "\ \ \#\#  $i=\(\$$i\)"
  done
}
...
showVals USER HOME PS1 outFile nflag
```

Doing glob matching

In bash and sh you must use `case`. Here is a handy function, `globmatch`, that lets you glob match anywhere.

```
  # Ex:  matches [ -q ] string globpattern
  # Does $1 match the glob expr $2 ?
  #     -q flag = set return status to 0 (true) or 1 (false)
  #  no -q flag = echo "1" (true) or "0" (false)
  # Unfortunately, the return status is opposite from the echo'ed string
globmatches () {
  if [ $1 = "-q" ]; then
    shift
    case "$1" in
      $2 ) true ;;
      * ) false ;;
    esac
  else
    case "$1" in
      $2 ) echo 1 ; true ;;
      * ) echo 0 ; false ;;
    esac
  fi
}

if globmatches -q $file "*.tar" ; then
  echo "Found a tar file"
elif globmatches -q $file "*zip" ; then
  echo "Found a zip file"
if
```

Extracting data

You will often get data with multiple fields or *words*. To extract and print the K-th word, where the first word is K=1, use either of

```
  set  - ... | echo $K          # purely shell based solution
  ... | awk '{ print $K; }'    # requires awk or nawk or gawk
  ... | cut -f K -s ' '        # least preferred method
```

For simple tasks, using the shell built in `set` is easiest. It is better to use awk (or gawk or nawk) because awk handles words separate by spaces and tabs correctly. The `cut` program (as of 2001) is quite stupid and assumes precisely one space between words.

To extract the K-th, M-th and P-th words, use either of

```
  awk '{ print $K, $M, $P; }'
  cut -f K,M,P -s ' '
```

Precede debugging/verbose messages with common prefix

I personally like '#' because this is the comment character for both scripts and perl. Sometimes, one shell script generates a second script, in which case I must precede optional messages with a comment character.

```
  echo "# Do not modify this script.  Auto-generated by master script $0"
  ...
  echo "# FYI, variable color=$color"
```

No full paths

Do not put full paths in your script, because if the path is wrong, say on a different OS/platform, you have to change all the paths in your script. Instead augment the PATH as necessary. E.g. if your script need to run `/usr/ucb/whoami`, then put the following in your script. On a different platform, you only have to augment the PATH differently.

```
PATH=/usr/ucb:$PATH
...
... whoami ...
```

Simple regular expression substitution

To change or *substitute* the text FROMX to TOX, use `sed`. You can specify regular expressions for FROMX. A

```
  sed -e "s/FROMX/TOX/"         # subst first occurrence
  sed -e "s/FROMX/TOX/g"        # subst all occurrences
    # strip off domain name (remove .in20pages.com)
  echo "speedster.in20pages.com" | sed -e "s/[.].*$//"
    # keep domain name (remove speedster.)
  echo "speedster.in20pages.com" | sed -e "s/^[^.]*[.]//"
```

Floating point math and base calculations

Use `dc`, the postfix or RPN calculator, or `bc` which takes human familiar infix notation. I strongly prefer `dc`. In the following examples, I store the results in variables `rx`, `rhex` and `wacko`. In `dc`, the commands `i`, `o`, `k` mean set the input base, output base, calculation precision, respectively. In `dc`, `p` means print the top of the stack.

```
    # calculate (2.718 + 1.414) / (3.141 - 2)  to 5 decimal places
  rx=`echo 5 k  2.718 1.414 + 3.141 2 - / p | dc`
    # convert 12345 to hex (base 16)
  rhex=`echo 16 o  12345 p | dc`
    # convert 12345 base 7 to octal (base 8) [All your base are belong to us]
  wacky=`echo 7 i 8 o  12345 p | dc`
```

One liners

- Use `pushd` and `popd` to change and restore the current directory. I usually redirect output to `/dev/null`.
- Use `mkdir -p` to create directories.
- Use `case` to do glob matching.

[LaTeX -> HTML by ltoh]
Russell W. Quong (*`ltoh@quong.REMOVE-THIS.SPAM.FILTER.PART-com.`*)
Last modified: `Jan 5 2005`