UNIX Shell Scripting

Ken Steube
UCSD Extension
steube@sdsc.edu

This course will teach you to write Bourne shell Scripts. We will then learn about C shell scripts, and will have a brief introduction to perl as well. Scripting skills have many applications, including:

- Ability to automate tasks, such as
  - Backups
  - Administration tasks
  - Periodic operations on a database via cron
  - Any repetetive operations on files
- Increase your general knowledge of UNIX
  - Use of environment
  - Use of UNIX utilities
  - Use of features such as pipes and I/O redirection

Our course syllabus is available on-line.

Some tips on working in SDSC's training lab are available.

I also have a comparison between features of the C shell and Bourne shell.

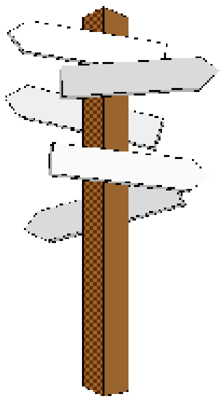
To connect to the SDSC lab computers from home or work you will need to use ssh instead of telnet or rlogin. Be sure to use an implementation of SSH that uses protocol version 1.0.

There is a free ssh program available from PuTTY.

SSH for UNIX is available from SSH.

For this class you will need a data file called last.out. To get this file, put the mouse over the blue text, press the right mouse button, and select "Save link as".

Table of Contents:



Section 1: Review of a few Basic UNIX Topics

Variables

- **Topics covered**: capturing output of a command
- **Utilities covered**: echo, expr

- Before trying the commands below start up a Bourne shell:

```
sh
```

- A variable stores a string

```
name="John Doe"
echo $name
```

- The quotes are required in the example above because the string contains a special character (the space)
- A variable may store a number

```
num=137
```

- The shell stores this as a string even though it appears to be a number
- A few UNIX utilities will convert this string into a number to perform arithmetic

```
expr $num + 3
```

- Try defining num as '7m8' and try the expr command again
- What happens when num is not a valid number?
- Now you may exit the Bourne shell with

```
exit
```

I/O Redirection

- **Topics covered**: specifying the input or capturing the output of a command
- **Utilities covered**: wc, sort

- The wc command counts the number of lines, words, and characters in a file

```
wc /etc/passwd
wc -l /etc/passwd
```

- You can save the output of wc (or any other command) with output redirection

```
wc /etc/passwd > wc.file
```

- You can specify the input with input redirection

```
wc < /etc/passwd
```

- Many UNIX commands allow this:

```
sort /etc/passwd
sort < /etc/passwd
```

- You can append lines to the end of an existing file with

```
wc -l /etc/passwd >> wc.file
```

Backquotes

- **Topics covered**: capturing output of a command
- **Utilities covered**: date

- The backquote character looks like the single quote or apostrophe, but slants the other way
- It is used to capture the output of a UNIX utility
- A command in backquotes is executed and then replaced by the output of the command
- Execute these commands

```
date
save_date=`date`
echo The date is $save_date
```

- Notice how echo prints the output of 'date'
- Store the following in a file named backquotes.sh and execute it

```
#!/bin/sh
# Illustrates using backquotes
# Output of 'date' stored in a variable
Today="`date`"
echo Today is $Today
```

- (To save the file, position the mouse over the the highlighted file name above and press the right mouse button for a menu. Select "Save link as".)
- Execute the script with

```
sh backquotes.sh
```

- Backquotes are very useful, but be aware that they slow down a script

Pipes

- **Topics covered**: using UNIX pipes
- **Utilities covered**: sort, cat, head

- Pipes are used for post-processing data
- One UNIX command prints results to the standard output (usually the screen), and another command reads that data and processes it

```
sort /etc/passwd | head -5
```

- Notice that this pipe can be simplified

```
cat /etc/passwd | head -5
```

- You could accomplish the same thing more efficiently with either of the two commands:

```
head -5 /etc/passwd
head -5 < /etc/passwd
```

awk

- **Topics covered**: processing columnar data
- **Utilities covered**: awk

- The awk utility is used for processing columns of data
- A simple example shows how to extract column 5 (the file size) from the output of ls -l

  ```
  ls -l | awk '{print $5}'
  ```

- A more complicated example shows how to sum the file sizes and print the result at the end of the awk run

  ```
  ls -al | awk '{sum = sum + $5} END {print sum}'
  ```

Section 2: Storing Frequently Used Commands in Files: Shell Scripts

Shell Scripts

- **Topics covered**: storing commands in a file and executing the file, $USER variable (standard Bourne shell variable)
- **Utilities covered**: date, cal, last, pipes

- Store the following in a file named simple.sh and execute it

  ```
  #!/bin/sh
  # Generate some useful info for
  # use at the start of the day
  date
  cal
  last $USER | head -6
  ```

- Shows current date, calendar, and a six of your previous logins for security check
- You might run this at the beginning of each day
- Notice that the commands themselves are not displayed, only the results
- To display the commands verbatim as they run, execute with

  ```
  sh -v simple.sh
  ```

- To echo the commands after variable translation, execute with

  ```
  sh -x simple.sh
  ```

- With -v or -x (or both) you can easily relate any error message that may appear to the command that generated it
- When an error occurs in a script, the script continues executing at the next command
- Verify this by changing 'cal' to 'caal' to force an error, and then run the script again
- Run the 'caal' script with 'sh -v simple.sh' and 'sh -x simple.sh' and verify the error message comes from cal
- Now you can re-use commands easily and save some typing and mistakes

- Other standard variable names include: HOME, PATH, TERM, PAGER, PRINTER

Page 6

Storing Strings in Variables

- **Topics covered**: variables store strings, creating and using variables
- **Utilities covered**: echo, ls, wc

- A variable is a name that stores a string
- Store the following in a file named variables.sh and execute it

```
#!/bin/sh
# An example with variables
filename="/etc/passwd"
echo "Check the permissions on $filename"
ls -l $filename
echo "Find out how many accounts there are on this system"
wc -l $filename
```

- Now if we change the value of $filename, the change is automatically propagated throughout the entire script

Page 7

Scripting With sed

- **Topics covered**: global search and replace, input and output redirection
- **Utilities covered**: sed

- Copy the file nlanr.txt to your home directory
- Change 'platform' to 'computer' with

```
sed -e 's/platform/computer/g' < nlanr.txt
```

- Save new text in a file with

```
sed -e 's/platform/computer/g' < nlanr.txt > nlanr.new
```

- Sed can do much more

Page 8

Section 3: More on Using UNIX Utilities

Performing Arithmetic

- **Topics covered**: integer arithmetic, preceding '*' with backslash to avoid file name wildcard expansion
- **Utilities covered**: expr

- Arithmetic is done with expr

```
expr 5 + 7
expr 5 \* 7
```

- Backslash required in front of '*' since it is a metacharacter and would be translated by the shell into a list of file names
- You can save arithmetic result in a variable
- Store the following in a file named arith.sh and execute it

```
#!/bin/sh
# Perform some arithmetic
x=24
y=4
Result=`expr $x \* $y`
echo "$x times $y is $Result"
```

Translating Characters

- **Topics covered**: converting one character to another, translating and saving string stored in a variable
- **Utilities covered**: tr

- Copy the file sdsc.txt to your home directory
- The utility tr translates characters

```
tr 'a' 'Z' < sdsc.txt
```

- This example shows how to translate the contents of a variable and display the result on the screen with tr
- Store the following in a file named tr1.sh and execute it

```
#!/bin/sh
# Translate the contents of a variable
Cat_name="Piewacket"
echo $Cat_name | tr 'a' 'i'
```

- This example shows how to change the contents of a variable
- Store the following in a file named tr2.sh and execute it

```
#!/bin/sh
# Illustrates how to change the contents of a variable with tr
Cat_name="Piewacket"
echo "Cat_name is $Cat_name"
Cat_name=`echo $Cat_name | tr 'a' 'i'`
echo "Cat_name has changed to $Cat_name"
```

- You can also specify ranges of characters.
- This example converts upper case to lower case

```
tr 'A-Z' 'a-z' < file
```

- Now you can change the value of the variable and your script has access to the new value

Section 4: Performing Search and Replace in Several Files

Processing Multiple Files

- **Topics covered**: executing a sequence of commands on each of several files with for loops
- **Utilities covered**: no new utilities

- Store the following in a file named loop1.sh and execute it

```
#!/bin/sh
# Execute ls and wc on each of several files
# File names listed explicitly
for filename in simple.sh variables.sh
do
        echo "Variable filename is set to $filename..."
        ls -l $filename
        wc -l $filename
done
```

- This executes three commands for each file name
- You should see three lines of output for each file name
- filename is a variable, set by "for" statement, referenced as $filename
- Now we can execute the same series of commands on each of several files

Page 11

Using File Name Wildcards in For Loops

- **Topics covered**: looping over files specified with wildcards
- **Utilities covered**: no new utilities

- Store the following in a file named loop2.sh and execute it

```
#!/bin/sh
# Execute ls and wc on each of several files
# File names listed using file name wildcards
for filename in *.sh
do
        echo "Variable filename is set to $filename..."
        ls -l $filename
        wc -l $filename
done
```

- Should see three lines of output for each file name ending in '.sh'
- Instead of listing files explicitly, uses file name wildcard to describe file names

Page 12

Search and Replace in Multiple Files

- **Topics covered**: combining for loops with utilities for global search and replace in several files
- **Utilities covered**: mv

- Store the following in a file named s-and-r.sh and execute it

```
#!/bin/sh
# Perform a global search and replace on each of several files
# File names listed explicitly
for text_file in sdsc.txt nlanr.txt
do
        echo "Editing file $text_file"
        sed -e 's/application/APPLICATION/g' $text_file > temp
        mv -f temp $text_file
done
```

- Sed cannot overwrite source file, so must use temp file + mv:
- Sed performs global search and replace
- Saves new result in file 'temp'
- Mv overwrites old file with new data

Section 5: Using Command-line Arguments for Flexibility

What's Lacking in the Scripts Above?

- **Topics covered**: looping over files specified with wildcards
- **Utilities covered**: no new utilities

- File names are hard-coded
- To execute for loops on different files, the user has to know how to edit the script
- Not simple enough for general use by the masses
- Wouldn't it be useful if we could easily specify different file names for each execution of a script?

What are Command-line Arguments?

- **Topics covered**: specifying command-line arguments
- **Utilities covered**: no new utilities

- Command-line arguments follow the name of a command

```
ls -l .cshrc /etc
```

- The command above has three command-line arguments

```
-l       (an option)
.cshrc   (a file name)
/etc     (a directory name)
```

- The argument '-l' will be interpreted as an option of ls

```
wc *.sh
```

- The command above has an unknown number of arguments, use 'echo *.sh' to see them
- Your scripts may also have arguments

Accessing Command-line Arguments

- **Topics covered**: accessing command-line arguments
- **Utilities covered**: no new utilities

- Store the following in a file named args1.sh and execute it

```
#!/bin/sh
# Illustrates using command-line arguments
# Execute with
#       sh args1.sh On the Waterfront
echo "First command-line argument is: $1"
echo "Third argument is: $3"
echo "Number of arguments is: $#"
echo "The entire list of arguments is: $*"
```

- Words after the script name are command-line arguments
- Arguments are usually file names

Looping Over the Command-line Arguments

- **Topics covered**: using command-line arguments in a for loop
- **Utilities covered**: no new utilities

- Store the following in a file named args2.sh and execute it

```
#!/bin/sh
# Loop over the command-line arguments
# Execute with
#       sh args2.sh simple.sh variables.sh
for filename in "$@"
do
        echo "Examining file $filename"
        wc -l $filename
done
```

- This script runs properly with any number of arguments, including zero
- Shorter form of the same loop

```
for filename
...
```

- Don't use

```
for filename in $*
```

- Fails if any arguments include spaces

Page 17

If Blocks

- **Topics covered**: testing conditions, executing commands conditionally
- **Utilities covered**: test (used by if to evaluate conditions)

- This will be covered on the whiteboard
- See Chapter 8 of the book

Page 18

The read Command

- **Topics covered**: reading a line from the standard input
- **Utilities covered**: no new utilities

- stdin is the keyboard unless input redirection used
- Read one line from stdin, store line in a variable

```
read variable_name
```

- Ask the user if he wants to exit the script
- Store the following in a file named read.sh and execute it

```
#!/bin/sh
# Shows how to read a line from stdin
echo "Would you like to exit this script now?"
read answer
if [ "$answer" = y ]
then
        echo "Exiting..."
        exit 0
fi
```

Page 19

Command Exit Status

- **Topics covered**: checking whether a command succeeds or not
- **Utilities covered**: no new utilities

- Every command in UNIX should return an exit status
- Status is in range 0-255
- Only 0 means success
- Other statuses indicate various types of failures
- Status does not print on screen, but is available thru variable $?
- Example shows how to examine exit status of a command
- Store the following in a file named exit-status.sh and execute it

```
#!/bin/sh
```

```
# Experiment with command exit status
echo "The next command should fail and return a status greater than zero"
ls /nosuchdirectory
echo "Status is $? from command: ls /nosuchdirectory"
echo "The next command should succeed and return a status equal to zero"
ls /tmp
echo "Status is $? from command: ls /tmp"
```

- Example shows if block using exit status to force exit on failure
- Store the following in a file named exit-status-test.sh and execute it

```
#!/bin/sh
# Use an if block to determine if a command succeeded
echo "This mkdir command fails unless you are root:"
mkdir /no_way
if [ "$?" -ne 0 ]
then
        # Complain and quit
        echo "Could not create directory /no_way...quitting"
        exit 1  # Set script's exit status to 1
fi
echo "Created directory /no_way"
```

- Exit status is $status in C shell

Regular Expressions

- **Topics covered**: search patterns for editors, grep, sed
- **Utilities covered**: no new utilities

- Zero or more characters: .*

```
grep 'provided.*access' sdsc.txt
sed -e 's/provided.*access/provided access/' sdsc.txt
```

- Search for text at beginning of line

```
grep '^the' sdsc.txt
```

- Search for text at the end of line

```
grep 'of$' sdsc.txt
```

- Asterisk means zero or more the the preceeding character

```
a*      zero or more a's
aa*     one or more a's
aaa*    two or more a's
```

- Delete all spaces at the ends of lines

```
sed -e 's/ *$//' sdsc.txt > sdsc.txt.new
```

- Turn each line into a shell comment

```
sed -e 's/^/# /' sdsc.txt
```

Greed and Eagerness

- Attributes of pattern matching
- Greed: a regular expression will match the largest possible string
- Execute this command and see how big a string gets replaced by an underscore

```
echo 'Big robot' | sed -e 's/i.*o/_/'
```

- Eagerness: a regular expression will find the first match if several are present in the line
- Execute this command and see whether 'big' or 'bag' is matched by the regular expression

```
echo 'big bag' | sed -e 's/b.g/___/'
```

- Contrast with this command (notice the extra 'g')

```
echo 'big bag' | sed -e 's/b.g/___/g'
```

- Explain what happens in the next example

```
echo 'black dog' | sed -e 's/a*/_/'
```

- Hint: a* matches zero or more a's, and there are many places where zero a's appear
- Try the example above with the extra 'g'

```
echo 'black dog' | sed -e 's/a*/_/g'
```

Regular Expressions Versus Wildcards

- **Topics covered**: clarify double meaning of asterisk in patterns
- **Utilities covered**: no new utilities

- Asterisk used in regular expressions for editors, grep, sed
- Different meaning in file name wildcards on command line and in find command and case statement (see below)

```
regexp  wildcard  meaning

.*      *         zero or more characters, any type
.       ?         exactly one character, any type
[aCg]   [aCg]     exactly one character, from list: aCg
```

- Regexps can be anchored to beginning/ending of line with ^ and $
- Wildcards automatically anchored to both extremes
- Can use wildcards un-anchored with asterisks

```
ls *bub*
```

Getting Clever With Regular Expressions

- **Topics covered**: manipulating text matched by a pattern
- **Utilities covered**: no new utilities

- Copy the file animals.txt to your home directory
- Try this sed command, which changes the first line of animals.txt

```
sed -e "s/big \(.*\) dog/small \1 cat/" animals.txt
```

- Bracketing part of a pattern with \(and \) labels that part as \1
- Bracketing additional parts of a pattern creates labels \2, \3, ...
- This sed command reverses the order of two words describing the rabbit

```
sed -e "s/Flopsy is a big \(.*\) \(.*\) rabbit/A big \2 \1 rabbit/" < animals.txt
```

The case Statement

- **Topics covered**: choosing which block of commands to execute based on value of a string
- **Utilities covered**: no new utilities

- The next example shows how to use a case statement to handle several contingencies
- The user is expected to type one of three words
- A different action is taken for each choice
- Store the following in a file named case1.sh and execute it

```
#!/bin/sh
# An example with the case statement
# Reads a command from the user and processes it
echo "Enter your command (who, list, or cal)"
read command
case "$command" in
        who)
                echo "Running who..."
                who
                ;;
        list)
                echo "Running ls..."
                ls
                ;;
        cal)
                echo "Running cal..."
                cal
                ;;
        *)
                echo "Bad command, your choices are: who, list, or cal"
                ;;
esac
exit 0
```

- The last case above is the default, which corresponds to an unrecognized entry
- The next example uses the first command-line arg instead of asking the user to type a command
- Store the following in a file named case2.sh and execute it

```
#!/bin/sh
# An example with the case statement
# Reads a command from the user and processes it
# Execute with one of
#       sh case2.sh who
#       sh case2.sh ls
#       sh case2.sh cal
echo "Took command from the argument list: '$1'"
case "$1" in
        who)
                echo "Running who..."
                who
                ;;
        list)
                echo "Running ls..."
                ls
                ;;
        cal)
                echo "Running cal..."
                cal
                ;;
        *)
                echo "Bad command, your choices are: who, list, or cal"
                ;;
esac
```

- The patterns in the case statement may use file name wildcards

The while Statement

- **Topics covered**: executing a series of commands as long as some condition is true
- **Utilities covered**: no new utilities

- The example below loops over two statements as long as the variable i is less than or equal to ten
- Store the following in a file named while1.sh and execute it

```
#!/bin/sh
# Illustrates implementing a counter with a while loop
# Notice how we increment the counter with expr in backquotes
i="1"
while [ $i -le 10 ]
do
        echo "i is $i"
        i=`expr $i + 1`
done
```

Example With a while Loop

- **Topics covered**: Using a while loop to read and process a file
- **Utilities covered**: no new utilities

- Copy the file while2.data to your home directory
- The example below uses a while loop to read an entire file
- The while loop exits when the read command returns false exit status (end of file)
- Store the following in a file named while2.sh and execute it

```
#!/bin/sh
# Illustrates use of a while loop to read a file
cat while2.data |   \
while read line
do
        echo "Found line: $line"
done
```

- The entire while loop reads its stdin from the pipe
- Each read command reads another line from the file coming from cat
- The entire while loop runs in a subshell because of the pipe
- Variable values set inside while loop not available after while loop

Interpreting Options With getopts Command

- **Topics covered**: Understand how getopts command works
- **Utilities covered**: getopts

- getopts is a standard UNIX utility used for our class in scripts getopts1.sh and getopts2.sh
- Its purpose is to help process command-line options (such as -h) inside a script
- It handles stacked options (such as -la) and options with arguments (such as -P used as -Pprinter-name in lpr command)
- This example will help you understand how getopts interprets options
- Store the following in a file named getopts1.sh and execute it

```
#!/bin/sh

# Execute with
#
#       sh getopts1.sh  -h  -Pxerox  file1  file2
#
# and notice how the information on all the options is displayed
#
# The string 'P:h' says that the option -P is a complex option
# requiring an argument, and that h is a simple option not requiring
# an argument.
#

# Experiment with getopts command
while getopts 'P:h' OPT_LETTER
do
        echo "getopts has set variable OPT_LETTER to '$OPT_LETTER'"
        echo "  OPTARG is '$OPTARG'"
done

used_up=`expr $OPTIND - 1`

echo "Shifting away the first \$OPTIND-1 = $used_up command-line arguments"

shift $used_up

echo "Remaining command-line arguments are '$*'"
```

- Look over the script
- getopts looks for command-line options
- For each option found, it sets three variables: OPT_LETTER, OPTARG, OPTIND
- OPT_LETTER is the letter, such as 'h' for option -h
- OPTARG is the argument to the option, such as -Pjunky has argument 'junky'
- OPTIND is a counter that determines how many of the command-line arguments were used up by getopts (see the shift command in the script)
- Execute it several times with

```
sh getopts1.sh -h -Pjunky
sh getopts1.sh -hPjunky
sh getopts1.sh -h -Pjunky /etc /tmp
```

- Notice how it interprets -h and gives you 'h' in variable OPT_LETTER
- Now you can easily implement some operation when -h is used
- Notice how the second execution uses stacked options
- Notice how the third execution examines the rest of the command-line after the options (these are usually file or directory names)

Example With getopts

- **Topics covered**: interpreting options in a script
- **Utilities covered**: getopts

- The second example shows how to use if blocks to take action for each option
- Store the following in a file named getopts2.sh and execute it

```
#!/bin/sh
#
# Usage:
#
#       getopts2.sh [-P string] [-h] [file1 file2 ...]
#
# Example runs:
#
#       getopts2.sh -h -Pxerox file1 file2
#       getopts2.sh -hPxerox file1 file2
#
# Will print out the options and file names given
#

# Initialize our variables so we don't inherit values
# from the environment
opt_P=''
opt_h=''

# Parse the command-line options
while getopts 'P:h' option
do
        case "$option" in
        "P")    opt_P="$OPTARG"
                ;;
        "h")    opt_h="1"
                ;;
        ?)      echo "getopts2.sh: Bad option specified...quitting"
                exit 1
                ;;
        esac
done

shift `expr $OPTIND - 1`

if [ "$opt_P" != "" ]
then
        echo "Option P used with argument '$opt_P'"
fi

if [ "$opt_h" != "" ]
then
        echo "Option h used"
fi

if [ "$*" != "" ]
then
        echo "Remaining command-line:"
        for arg in "$@"
        do
                echo "  $arg"
        done
fi
```

- Execute it several times with

```
sh getopts2.sh -h -Pjunky
sh getopts2.sh -hPjunky
sh getopts2.sh -h -Pjunky /etc /tmp
```

- Can also implement actions inside case statement if desired

Section 6: Using Functions

Functions

- Sequence of statements that can be called anywhere in script
- Used for
  - Good organization
  - Create re-usable sequences of commands

Define a Function

- Define a function

```
echo_it () {
  echo "In function echo_it"
}
```

- Use it like any other command

```
echo_it
```

- Put these four lines in a script and execute it

Function Arguments

- Functions can have command-line arguments

```
echo_it () {
  echo "Argument 1 is $1"
  echo "Argument 2 is $2"
}
echo_it arg1 arg2
```

- When you execute the script above, you should see

```
Argument 1 is arg1
Argument 2 is arg2
```

- Create a script 'difference.sh' with the following lines:

```
#!/bin/sh
echo_it () {
        echo Function argument 1 is $1
}
echo Script argument 1 is $1
```

```
echo_it Barney
```

- Execute this script using

```
sh difference.sh Fred
```

- Notice that '$1' is echoed twice with different values
- The function has separate command-line arguments from the script's

Example With Functions

- Use functions to organize script

```
read_inputs () { ... }
compute_results () { ... }
print_results () { ... }
```

- Main program very readable

```
read_inputs
compute_results
print_results
```

Functions in Pipes

- Can use a function in a pipe

```
ls_sorter () {
  sort -n +4
}
ls -al | ls_sorter
```

- Function in pipe executed in new shell
- New variables forgotten when function exits

Inherited Variables

- Variables defined before calling script available to script

```
func_y () {
  echo "A is $A"
  return 7
}
A='bub'
func_y
if [ $? -eq 7 ] ; then ...
```

- Try it: is a variable defined inside a function available to the main program?

Functions -vs- Scripts

- Functions are like separate scripts
- Both functions and scripts can:
- Use command-line arguments

```
echo First arg is $1
```

- Operate in pipes

```
echo "test string" | ls_sorter
```

- Return exit status

```
func_y arg1 arg2
if [ $? -ne 0 ] ...
```

Libraries of Functions

- Common to store definitions of favorite functions in a file
- Then execute file with

```
. file
```

- Period command executes file in current shell
- Compare to C shell's source command

Section 7: Miscellaneous

Here Files

- Data contained within script

```
cat << END
This script backs up the directory
named as the first command-line argument,
which in your case in $1.
END
```

- Terminator string must begin in column one
- Variables and backquotes translated in data
- Turn off translation with \END

Page 38

Example With Here File

- Send e-mail to each of several users

```
for name in $USER
do
  mailx -s 'hi there' $name << EOF
  Hi $name, meet me at the water
  fountain
EOF
done
```

- Use <<- to remove initial tabs automatically

Page 39

Set: Shell Options

- Can change Bourne shell's options at runtime
- Use set command inside script

```
set -v
set +v
set -xv
```

- Toggle verbose mode on and off to reduce amount of debugging output

Page 40

Set: Split a Line

- Can change Bourne shell's options

```
set -- word1 word2
echo $1, $2
  word1, word2
```

- Double dash important!
- Word1 may begin with a dash, what if word1 is '-x'?
- Double dash says "even if first word begins with '-', do not treat it as an option to the shell

Page 41

Example With Set

- Read a line from keyboard

- Echo words 3 and 5

```
read var
set -- $var
echo $3 $5
```

- Best way to split a line into words

Section 8: Trapping Signals

What are Signals?

- Signals are small messages sent to a process
- Process interrupted to handle signal
- Possibilities for managing signal:
    - Terminate
    - Ignore
    - Perform a programmer-defined action

Common Signals

- Common signals are
    - SIGINTR sent to foreground process by ^C
    - SIGHUP sent when modem line gets hung up
    - SIGTERM sent by kill -9
- Signals have numeric equivalents

```
2 SIGINTR
9 SIGTERM
```

Send a Signal

- Send a signal to a process

```
kill -2 PID
kill -INTR PID
```

Trap Signals

- Handling Signals

```
trap "echo Interrupted; exit 2" 2
```

- Ignoring Signals

```
trap "" 2 3
```

- Restoring Default Handler

```
trap 2
```

Where to Find List of Signals

- See file

```
/usr/include/sys/signal.h
```

User Signals

- SIGUSR1, SIGUSR2 are for your use
- Send to a process with

```
kill -USR1 PID
```

- Default action is to terminate process

Experiment With Signals

- Script that catches USR1
- Echo message upon each signal

```
trap 'echo USR1' 16
while : ; do
  date
  sleep 3
done
```

- Try it: does signal interrupt sleep?

Section 9: Understanding Command Translation

Command Translation

- Common translations include
    - Splitting at spaces, obey quotes
    - $HOME -> /users/us/freddy
    - `command` -> output of command
    - I/O redirection
    - File name wildcard expansion
- Combinations of quotes and metacharacters confusing
- Resolve problems by understanding order of translations

Experiment With Translation

- Try wildcards in echo command

```
echo b*
b budget bzzzzz
```

- b* translated by sh before echo runs
- When echo runs it sees

```
echo b budget bzzzzz
```

- Echo command need not understand wildcards!

Order of Translations

- Splits into words at spaces and tabs
- Divides commands at

```
; & | && || (...) {...}
```

- Echos command if -v
- Interprets quotes
- Performs variable substitution

Order of Translations (continued)

- Performs command substitution
- Implements I/O redirection and removes redirection characters
- Divides command again according to IFS
- Expands file name wildcards
- Echos translated command if -x
- Executes command

Exceptional Case

- Delayed expansion for variable assignments

```
VAR=b*
echo $VAR
  b  b_file
```

- Wildcard re-expanded for each echo

Examples With Translation

- Variables translated before execution
- Can store command name in variable

```
command="ls"
$command
  file1 file2 dir1 dir2...
```

- Variables translated before I/O redirection

```
tempfile="/tmp/scriptname_$$"
ls -al > $tempfile
```

Examples (continued)

- Delayed expansion of wildcards in variable assignment
- Output of this echo command changes when directory contents change (* is re-evaluated each time the command is run)

```
x=*
echo $x
```

- Can view values stored in variables with

```
set
```

- Try it: verify that the wildcard is stored in x without expansion

Page 56

Examples (continued)

- Wildcards expanded after redirection (assuming file* matches exactly one file):

```
cat < file*
  file*: No such file or directory
```

- Command in backquotes expanded fully (and before I/O redirection)

```
cat < `echo file*`
  (contents of file sent to screen)
```

Page 57

Eval Command

- Forces an extra evaluation of command

```
eval cat \< file*
  (contents of matching file)
```

- Backslash delays translation of < until second translation

Page 58

Section 10: Writing Advanced Loops

While loops

- Execute statements while a condition is true

```
i=0
while [ $i -lt 10 ]
do
  echo I is $i
  i=`expr $i + 1`
done
```

Page 59

Until loops

- Execute statements as long as a condition is false

```
until grep "sort" dbase_log > /dev/null
do
  sleep 10
done
echo "Database has been sorted"
```

- Example executes until grep is unsuccessful

Redirection of Loops

- Can redirect output of a loop

```
for f in *.c
do
  wc -l $f
done > loop.out
```

- Loop runs in separate shell
- New variables forgotten after loop
- Backgrounding OK, too

Continue Command

- Used in for, while, and until loops
- Skip remaining statements
- Return to top of loop

```
for name in *
do
  if [ ! -f $name ] ; then
    continue
  fi
  echo "Found file $name"
done
```

- Example loops over files, skips directories

Break Command

- Used in for, while, and until loops
- Skip remaining statements
- Exit loop

```
for name in *
do
  if [ ! -r $name ] ; then
    echo "Cannot read $name, quitting loop"
    break
  fi
  echo "Found file or directory $name"
done
```

- Example loops over files and directories, quits if one is not readable

Case Command

- Execute one of several blocks of commands

```
case "string" in
pattern1)
  commands ;;
pattern2)
  commands ;;
*) # Default case
  commands ;;
esac
```

- Patterns specified with file name wildcards

```
quit) ...
qu*)   ...
```

Example With Case

- Read commands from keyboard and interpret
- Enter this script 'case.sh'

```
echo Enter a command
while read cmd
do
  case "$cmd" in
    list) ls -al ;;
    freespace) df . ;;
    quit|Quit) break ;;
    *) echo "$cmd: No such command" ;;
  esac
done
echo "All done"
```

- When you run it, the script waits for you to type one of:

```
list
freespace
quit
Quit
```

- Try it: modify the example so any command beginning with characters "free" runs df

Infinite Loops

- Infinite loop with while

```
while :
do
  ...
done
```

- : is no-op, always returns success status
- Must use break or exit inside loop for it to terminate

Section 11: Forking Remote Shells

Remote Shells

- Rsh command

```
rsh hostname "commands"
```

- Runs commands on remote system
- Must have .rhosts set up
- Can specify different login name

```
rsh -l name hostname "commands"
```

Examples With rsh

- Check who's logged on

```
rsh spooky "finger"
```

- Run several remote commands

```
rsh spooky "uname -a; time"
```

- Executes .cshrc on remote system
- Be sure to set path in .cshrc instead of .login

Access Control with .Rhosts

- May get "permission denied" error from rsh
- Fix this with ~/.rhosts on remote system
- Example: provide for remote shell from spunky to spooky

```
spunky % rlogin spooky
spooky % vi ~/.rhosts
        (insert "spunky login-name")
spooky % chmod 600 ~/.rhosts
spooky % logout
spunky % rsh spooky uname -a
        spooky 5.5 sparc SUNW,Ultra-1
```

- May also rlogin without password: security problem!

Remote Shell I/O

- Standard output sent to local host

```
rsh spooky finger > finger.spooky
```

- Standard input sent to remote host

```
cat local-file | rsh spooky lpr -
```

Return Status

- Get return status of rsh

```
rsh mayer "uname -a"
echo $?
```

- Returns 0 if rsh managed to connect to remote host
- Returns 1 otherwise
    - Invalid hostname
    - Permission denied

Remote Return Status

- What about exit status of remote command?
- Have to determine success or failure from stdout or stderr

Section 12: More Miscellaneous

Temporary Files

- Use unique names to avoid clashes

```
tempfile=$HOME/Weq_$$
command > $tempfile
```

- $$ is PID of current shell
- Avoids conflict with concurrent executions of script
- Do not use /tmp!

Page 73

Wait Command

- Wait for termination of background job

```
command &
pid=$!
(other processing)
wait $pid
```

- Allows overlap of two or more operations

Page 74

Section 13: Using Quotes

Quotes

- Provide control of collapsing of spaces and translation of variables
- Try it: run three examples
- No quotes (variables translated, spaces collapsed)

```
echo Home:   $HOME
  Home: /users/us/freddy
```

- Double quotes (no collapsing)

```
echo "Home:   $HOME"
```

```
  Home:   /users/us/freddy
```

- Single quotes (no translation or collapsing)

```
echo 'Home:   $HOME'
  Home:   $HOME
```

- Try it: single quotes within double quotes

```
echo "Home directory '$HOME' is full..."
```

Metacharacters

- Characters with special meaning to shell

```
" ' ` $ * [ ] ?
; > < & ( ) \
```

- Avoid special meaning with quoting

```
echo 'You have $20'
```

- Backslash like single quotes
- Applies only to next character

```
echo You have \$20
```

Examples With Quotes

- Bad command line:

```
grep dog.*cat file
```

- Shell tries to expand dot.*cat as file name wildcard
- Use quotes to avoid translation

```
grep 'dog.*cat' file
```

- Single quotes OK in this case because we don't need variable translation

More Examples With Quotes

- Read name and search file for name

```
read name
grep "$name" dbase
```

- Single quotes not OK because we need variable translation

Searching for Metacharacters

- Bad command line: search for dollar sign

```
grep "Gimme.*$20" file
```

- Problem: shell translates variable $20
- Solution: use single quotes

```
grep 'Gimme.*$20' file
```

Last modified: 5/23/2008 6:59:11 PM

Last modified: Wed 15 Aug 2007 11:17:36 AM EDT

Last modified: Tue 10 Dec 2002 06:48:21 AM EST