SYST13416
Introduction to Linux
Operating

The Shell

CONTROL STRUCTURES

Objectives

- Control structures: sequence, iteration, and decision
- Structures: for, select, case, if, while, until
- Discuss instances where scripts are useful
- Write, test, and debug scripts to automate common tasks
- Control structures (decisions: if, case; iteration: while, for; sequence: when to use semi-colon)
- Variables (user-defined, shell, environment)
- Operators (assignment, arithmetic, conditional, file)
- Command line arguments ($0, $n, $#)
- Commands: shift, let, read, echo

Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems
by Dave Taylor
No Starch Press © 2004 (448 pages)
This cookbook of useful, customizable, and fun scripts gives you the tools to solve common Linux, Mac OS X and UNIX problems and personalize your computing environment.

Note:
In the labs the symbol » may be used to represent the command prompt.

Note the indentation to make the code easier to read.

Control structures Definitions

Sequence

- Most commands are executed in a sequence, usually each line contains a single command. The interpreter execute the command, when it finishes, goes to the next line, and executes the next command. If you want to put more than one command on a single line, you need to separate the commands with a semi-colon (;). The execution is exactly the same. Usually we do this to save space, but keep in mind that readability of the code decreases.

Decision

- Sometimes, we need to make a decision or ask a question, and depending on the answer, we take a different path. All decisions are binary using the if...then...else...fi structure. When you making more than one decision, you nest the decision, creating a unique path for each combination of choices using nested if or case.

Iteration

- Often, we need to repeat the same set of commands a number of times, which we do by putting the block of code inside a loop. The control structures for loops to examine in the following examples are while, for, and foreach.

Control Structures Overview

(reference the bash manual page)

for *name* [**in** *word*] ; **do** *list* ; **done**

- The list of words following **in** is expanded, generating a list of items. The variable *name* is set to each element of this list in turn, and *list* is executed each time. The return status is the exit status of the last command that executes. If the expansion of the items following **in** results in an empty list, no commands are executed, and the return status is 0.

for ((*expr1* ; *expr2* ; *expr3*)) ; **do** *list* ; **done**

- First, the arithmetic expression *expr1* is evaluated according to the rules described under **ARITHMETIC EVALUATION**. The arithmetic expression *expr2* is then evaluated repeatedly until it evaluates to zero. Each time *expr2* evaluates to a non-zero value, *list* is executed and the arithmetic expression *expr3* is evaluated. If any expression is omitted, it behaves as if it evaluates to 1. The return value is the exit status of the last command in *list* that is executed, or false if any of the expressions is invalid.

select *name* [**in** *word*] ; **do** *list* ; **done**

- The list of words following **in** is expanded, generating a list of items. The set of expanded words is printed on the standard error, each preceded by a number. If the **in** *word* is omitted, the positional parameters are printed. The **PS3** prompt is then displayed and a line read from the standard input. If the line consists of a number corresponding to one of the displayed words, then the value of *name* is set to that word. If the line is empty, the words and prompt are displayed again. If EOF is read, the command completes. Any other value read causes *name* to be set to null. The line read is saved in the variable **REPLY**. The *list* is executed after each selection until a **break** command is executed. The exit status of **select** is the exit status of the last command executed in *list*, or zero if no commands were executed.

case *word* **in** [[(] *pattern* [**|** *pattern*]

- A **case** command first expands *word*, and tries to match it against each *pattern* in turn, using the same matching rules as for pathname expansion. The *word* is expanded using tilde expansion, parameter and variable expansion, arithmetic substitution, command substitution, process substitution and quote removal. Each *pattern* examined is expanded using tilde expansion, parameter and variable expansion, arithmetic substitution, command substitution, and process substitution. If the shell option **nocasematch** is enabled, the match is performed without regard to the case of alphabetic characters. When a match is found, the corresponding *list* is executed. If the **;;** operator is used, no subsequent matches are attempted after the first pattern match. Using **;&** in place of **;;** causes execution to continue with the *list* associated with the next set of patterns. Using **;;&** in place of **;;** causes the shell to test the next pattern list in the statement, if any, and execute any associated *list* on a successful match. The exit status is zero if no pattern matches. Otherwise, it is the exit status of the last command executed in *list*.

if *list*; **then** *list;*
[**elif** *list*; **then** *list*;] ... [**else** *list*;] **fi**

- The **if** *list* is executed. If its exit status is zero, the **then** *list* is executed. Otherwise, each **elif** *list* is executed in turn, and if its exit status is zero, the corresponding **then** *list* is executed and the command completes. Otherwise, the **else** *list* is executed, if present. The exit status is the exit status of the last command executed, or zero if no condition tested true.

while *list*; **do** *list*; **done**
until *list*; **do** *list*; **done**

- The **while** command continuously executes the **do** *list* as long as the last command in *list* returns an exit status of zero. The **until** command is identical to the **while** command, except that the test is negated; the **do** *list* is executed as long as the last command in *list* returns a non-zero exit status. The exit status of the **while** and **until** commands is the exit status of the last **do** *list* command executed, or zero if none was executed.

Examples

Structure: if…then…else…fi

- Create a script that will find out whether the user is a full time student. If the user enters 'Y', direct them to go to building J, otherwise send them to building A. Which control structure is most suitable to use for this problem?

```
#!/bin/sh
echo -n "Are you a full-time student?"
read choice
if [ $choice = Y ]
then
    echo "Please go to Building J"
else
    echo "Please go to Building A"
fi
```

Structure: nested if…then…else…fi

- Write a shell script whose single command line argument is a file. If you run the program with an ordinary file, the program displays the owner's name and last update time for the file. If the program is run with more than one argument, it generates meaningful error messages.

```
#!/bin/sh
if [ $# = 1 ]
then
   if [ -f  $1 ]
   then
          echo –n "The owner and last update time for $1 is "
          ls -l $1 | tr –s ' ' | cut –d' ' -f3,8
   else
          echo "$1 is not an ordinary file"
   fi
else
   echo "Please enter one argument, a file name."
fi
```


Structure: for…in; Using a List

- Create and examine the following script . Note the use of the for control structure.

```
#!/bin/sh
echo "The things I like to do include:"
for myHobbies in read, ski, "collect stamps", run
do
    echo $myHobbies
done
```

Structure: for…in; Command Substitution

- The list membership can come from an array or a text file. In the following example, create a text file that contains a number of lines, each line consisting of a name of an animal and name the file *animalList*.

```
#!/bin/sh
echo "You can see the following animals in our zoo: "
for animal in `cat animalList`
do
   echo $animal
done
```

Structure: while…do…done; Shift Operator

- Create a script with the following content. Note the use of the while loop. Remember while loop is top-checking, meaning that it will enter the loop only if the condition is true. What does the script do? What does $# and $1 do? What does the shift operator do? Hint: What do you need to provide on the command line after the script name.

```
#!/bin/sh
while [ $# -gt 0 ]
do
   echo $1
   shift
done
```

Structure: C-style for…do…done

- Create a script to illustrate the C-style for loop. The First example **initialize**s the variable x to 0 (arithmetic expression), the arithmetic expression $x<5 (**condition**: Is value of x less than 5?) is then evaluated repeatedly until it evaluates to zero. Each time *it* evaluates to a non-zero value, *the do-done block* is executed and the arithmetic expression *x=$x+1* is evaluated (in other words, the value of x is **increment**ed, or changed, each time the body of the loop is executed. This is equivalent to a while loop as demonstrated in Example 2.

```
#!/bin/bash
# Examples using the C-style for loop
echo; echo Example 1
for (( x=0; $x<5; x=$x+1 ))
do
        echo -n $x" "
done

echo; echo Example 2
let x=0
while [ $x -lt 5 ]; then
do
    echo –n $x” “
    let x=$x+1
done

echo; echo Example 3
for (( y=25; $y > 0; y-- ))
do
        echo -n $y" "
done

echo; echo Example 4
for (( z=23; $z >= 0; z=$z-2 ))
do
        echo -n $z" "
done
```


Structure: if, while; OR operator

- In this example, note the structure if...then...else...fi nested inside a while…do…done structure and the use of the logical or operator:

```
 1 #!/bin/sh
 2 #Name: Ellen
 3 #Purpose: Demonstrate the OR operator
 4 repeat=yes
 5 input=yes
 6 while [ $repeat = yes ]
 7 do
 8    echo "What is your name?"
 9    read name
10    if [ $name = "Ellen" -o $name = "ellen" -o $name = "ELLEN" ]
11    then
12          echo "Hello, $name"
13    else
14          echo "Hello, $name. You are not Ellen"
15    fi
16    echo "Again? (To stop, type: no)  "
17    read input
18    if [ $input = no ]
19    then
20          repeat=no
21    fi
22 done
```

DATA STRUCTURES

Objectives

- Data Structure: **array**

Data Structures

Arrays

- Arrays are variables that store an *ordered list* of *scalar values* that are accessed with numeric subscripts
- Korn and Bash support one-dimensional arrays; the first element has index 0.
- Bash supports very, very large arrays. Korn arrays hold up to 1024 elements.

Processing Arrays

- Array elements need not to be assigned contiguously.
- Variable is declared as an array using one of the commands:
```
declare –a array_name
local –a array_name
readonly –a array_name
```
- Array variable can be initialized at the declaration time using syntax:
```
name=(value0 value1 … valueN)
```
where valueN is of form [[subscript]=]string.
- A particular element of an array can be assigned using expression:
```
arrayName[subscript]=value
```
- An array element is referenced by using
```
${arrayName[subscript]}
```
- Example:
```
fruit=(apple pear orange banana kiwi)
echo ${fruit[2]}
orange
```
- The size (in bytes) of an array can be displayed using
```
${#arrayName[subscript]}
```

- If no subscript is used, the size of the first element is displayed.
  `echo ${#arrayName}`
- If * is used as a subscript, the number of array elements is displayed.
  `echo ${arrayName[*]}`
- To display the size of the first element in the **fruit** array (which is *apple*):
  `echo ${#fruit}`
- To display the first element of the **fruit** array (which is *apple*):
  `echo ${#fruit[*]}`
- To `display` the entire array **fruit** (which is *apple pear orange banana kiwi*`)`
  `echo ${fruit[*]}`

- Assign a value to an individual element:
  `Month[1]=31`
  `Month[6]=30`
  `To evaluate an individual`

Array Example 1

```
array=(red green blue yellow magenta)
len=${#array[*]}
echo "The array has $len members. They are:"
i=0
while [ $i -lt $len ]; do
    echo "$i: ${array[$i]}"
    let i++
done
```

Array Example 2

```
array=(`ls`)
len=${#array[*]}
echo "The array has $len members. They are:"
i=0
while [ $i -lt $len ]; do
    echo "$i: ${array[$i]}"
    let i++
done
```


Comparing [ ] and test evaluation

1. Consider the following script fragment:
   ```
   echo -n "Try to guess my favourite color: "
   read guess
   while  [  "$guess" !=  "red"  ];  do
      echo "No, not that one. Try again.  "; read guess
   done
   ```
   Change the line that reads:
   `while  [  "$guess" !=  "red"  ];  do`
   to this:
   `while test $guess != "red" ; do`
   Examine the effect of the change.

ADVANCED SCRIPTS

Objectives

- Command: **read**
- Command: **tput**
- Command: **shift**
- Command: **clear**
- Copying user input
- Creating menus

Advanced Scripts

The read command

- The read command reads one line from the standard input and assigns the line to one or more variables.
- If you enter more words than read has variables, read assigns one word to each variable, with all the left-over words going to the last variable.
- If input contains special characters, and you want them to be just values of variables, use double quotes.
- If you want the shell to use the special meanings of the special characters, do not use quotation marks.

```
cat > readx
echo –n ”Enter a command: ”
read command
$command
echo Thanks
Ctrl+D

readx
Enter a command: who
alex tty11 Jun 17 07:09
scott tty7 Jun 17 08:23
Thanks

cat > readc
echo –n ”Enter something: ”
read word1 word2
echo ”Word 1 is: $word1”
echo ”Word 2 is: $word2”
Ctrl+D
```

```
readc
Enter something: Monty Python's Flying Circus
Word 1 is: Monty
Word 2 is: Python's Flying Circus
```

- When you enclose a command between two backquotes (`), the shell replaces the command with the output of the command (command substitution).

```
cat > dir
echo You are using the `pwd` directory.
Ctrl+D
dir
```

The output will look something like:

```
You are using the /home/jenny directory.
```


The **tput** command

Using the tput Command you can manage the layout of the screen:

- The tput command is used to place the prompt (cursor) at the user's data-entry point on the screen
- The tput command initializes the terminal to respond to a setting that the user chooses
- Move the cursor to row 0, column 0, the upper-left corner
  tput cup 0 0
- Clear the screen
  tput clear
- Print the number of columns for the current terminal
  tput cols
- Set boldfaced type
  bold=`tput smso `
  offbold=`tput rmso`

The **shift** command:

Shifting positional parameters left

- Many programs use a loop to iterate through its arguments but without including the first argument. This argument could represent a directory, and the remaining could be ordinary files. This is what shift does this. Each call to shift transfers the contents of a positional parameter to its immediate lower numbered one. $2 becomes $1, $3 becomes $2, and so on.

The **clear** command

Clearing the screen

- To clear the screen at the prompt, use the command clear. Note that the tput command also takes an argument called clear, which does the same thing.

Creating menus

Coping with user input

Coping with user input

- Here is an example that relies on user input to decide what to do. It exploits a shell feature as an easy way to create a menu of choices:

```
PS3="Choose (1-5):"
echo "Choose from the list below."
select name in red green blue yellow magenta
do
    break
done
echo "You chose $name.
```

- When run, it looks like this:

```
$ ./myscript.sh
```

- Output:

```
Choose from the list below.
1) red
2) green
3) blue
4) yellow
5) magenta
Choose (1-5):4
You chose yellow.
```


Interaction with the user

The following script capture all other user entries and display appropriate message. After this message is displayed for a short interval, the menu should refresh and the user should be able to select the options again. This script will require the use of two control structures, a while structure and a case structures.

```
#!/bin/sh
leave=no
while [ $leave = no ]; do
      sleep 3; clear; echo
      echo "$LOGNAME's menu"; echo
      echo "a- Display today's date and time."
      echo "b- Display current working directory."
      echo "x- Exit."
      echo -n "Please enter your selection $LOGNAME: "
      read selection; echo
      case $selection in
            a|A)
                  echo -n "The date and time is: "; date;;
            b|B)
                  echo -n "Your current working directory is: "
                  pwd ;;
            x|X)
                  leave=yes;;
            *)
                  echo "Your choice was not understood.  Try again!"
            esac
done
```

- Add the following menu items to the previous script:
  - c) List long directory listing that includes hidden files
  - d) Display who is logged on the system in alphabetical order
  - e) Display current month's calendar
  - f) Display files the user selects
  - g) Move files that the user selects to the 'garbage' directory

Change the script to display the menu starting on row 7 and column 30. Highlight (bold) the menu's title.

EXAMPLE SCRIPTS

Objectives

- Bash Shell Scripting

Example 11-A: The or operator

Create, debug, and test the following script. Name it **or.sh**

Update and add any inline documentation that will help you remember what specific commands and the script do. Note any changes that you have made to the file and why.

Can you suggest alternative ways of handling the case sensitivity?

```
 1 #!/bin/bash
 2 #Name: SYST13416
 3 #Purpose: Demonstrate the OR operator and the
             if…then…else…fi control structure
 4 repeat=yes
 5 input=yes
 6 while [ $repeat = yes ]
 7 do
 8    echo "What is your name?"
 9    read name
10    if [ $name = "Ellen" -o $name = "ellen"
   -o $name = "ELLEN" ]
11    then
12       echo "Hello, $name"
13    else
14       echo "Hello, $name. You are not Ellen"
15    fi
16    echo "Again? (To stop, type: no)  "
17    read input
18    if [ $input = no ]
19    then
20       repeat=no
21    fi
22 done
```

Example 11-B
Doing a little arithmetic?

Create, debug, and test the following script. Name it **countup.sh**

Update and add any inline documentation that will help you remember what specific commands and the script do. Note any changes that you have made to the file and why.

```
#!/bin/bash
#ASSIGNMENT: Example
#PROGRAM NAME: countup.sh
#AUTHOR: SYST13416
#DATE CREATED: Winter 2004
#DATE UPDATED: Winter 2005
#PURPOSE: Demonstration of using while structure to count up
#         from one to the number entered as the first argument.
#         This if statement checks that there is one argument and
#         displays command usage to the user.
if [ $# != 1 ]; then
       echo "Usage: $0 integerArgument"
       exit 1
fi
#The following while loop counts up
target="$1"; current=1; sum=0; count=0
while [ $current -le $target ]; do
       echo -n "$current "
       sum=`expr $sum + $current`
       current=$(( current+1 ))
       if [ $? != 0 ]; then
            exit 1
       fi
       count=$(( count+1 ))
       shift
done
#The following segment calculates the average and displays the sum and
#average
if [ $count != 0 ]; then
       let average=$sum/$count
else
       echo "The sum of the given $count numbers is $sum."
       echo "Average cannot be calculated"
       exit 1
fi
echo
echo "The sum of the given $count numbers is $sum. The average is $average."
exit 0
```


Example 11-C
Script to safely remove files

Create, debug, and test the following script. Name it **rmv** (no extension). Update and add any inline documentation that will help you remember what specific commands and the script do. Note any changes that you have made to the file and why.

```
 1 #!/bin/bash
 2 # rmv - a safe delete program
 3 # uses a trash directory under your home directory
 4 mkdir $HOME/.trash 2>/dev/null
 5 cmdlnopts=false
 6 delete=false
 7 empty=false
 8 list=false
 9 # The script uses the bash shell getopts command to look at your command
10 # line for any options. If a matching letter is found by the case
11 # statement, the script commands up until the two semicolons are executed.
12 while getopts “dehl” cmdlnopts; do
13     case “$cmdlnopts” in
14           d ) /bin/echo “deleting: \c” $2 $3 $4 $5 ; delete=true ;;
15           e ) /bin/echo “emptying the trash...” ; empty=true ;;
16           h ) /bin/echo “safe file delete v1.0”
17              /bin/echo “rmv -d[elete] -e[mpty] -h[elp] -l[ist] file1-4” ;;
18           l ) /bin/echo “your .trash directory contains:” ; list=true ;;
19     esac
20 done
21 if [ $delete = true ]; then
22     mv $2 $3 $4 $5 $HOME/.trash
23     /bin/echo “rmv finished.”
24 fi
25 if [ $empty = true ]; then
26     /bin/echo “empty the trash? \c”
27     read answer
28     case “$answer” in
29 #          f) rm -fr $HOME/.trash/* ;;
30           y) rm -i $HOME/.trash/* ;;
31           n) /bin/echo “trashcan delete aborted.” ;;
32     esac
33 fi
34 if [ $list = true ]; then
35     ls -l $HOME/.trash
36 fi
```

Contents