Lecture 8

Shell Programming – Control Constructs

COP 3353 Introduction to UNIX

*

Command Line Arguments Cont.

- $# contains the number of command line arguments.
- $@ will be replaced by a string containing the command line arguments
- Example script echo.sh

```
#!/bin/sh
echo "The" $# "arguments entered:" $@
```

- Usage:

```
echo.sh alpha beta gamma
```

- Output:

```
The 3 arguments entered: alpha beta gamma
```

Testing Conditions

- There are two ways to test for conditions. The two general forms are:

    test <condition>

or

    [<condition>]

- The latter method is easier to read. ***Remember to include a space before and after the bracket
- A condition can be reversed with a ! before the condition (this is the same as “not” some condition)

    [!<condition>]

- A ‘:’ command in place of condition always returns true

*

Testing File Attributes - examples

- To test if a file is readable

[-r prog.txt] #tests if prog.txt is readable

[-r $1.c] #tests if <argument1>.c is readable

- To test if a file is writeable

[-w specialfile.txt] #tests if specialfile.txt is writable

- To test if a file is executable

[-x prog4.sh] #tests if prog4.sh is executable

- To test if a file exists

[-f temp.text] #to test if temp.txt is a file (exists)

- Testing for the negation - use ! (eg. not writeable)

[! -w nochange.txt] #tests if nochange.txt is NOT writable

Numeric Tests

- The following operators can be used for *numeric* tests:

{ -eq, -ne, -gt, -ge, -lt, -le }

(equal, not equal, greater than, greater than or equal, less than, less than or equal. CANNOT USE >, <, >=, <=, == for numbers)

- Examples

[$1 –lt $2] #tests if argument 1 is less than argument 2

[$1 –gt 0] #tests if argument 1 is greater than 0

[$# -eq 2] #tests if the number of arguments is equal to 2

[$# -lt 3] #tests if the number of arguments is less than 3

*

Simple If Statement

- General Form:

if <condition>
then
    one-or more commands
fi

- Example:

if [-r tmp.text]
then
    echo “temp.text is a readable file”
fi

General If Statement

- General form:

if <condition> #BEGINS the if block
then
    one-or-more-commands
elif <condition>
then
    one-or-more-commands
… #any number of elif clauses (else if)
else
    one-or-more-commands
fi #CLOSES the entire if block

- Note that you can have 0 or more elif statements and that the else is optional.

*

If Statement Example, using elif, else

```
if [ $var1 –lt $var2 ]
then
        echo $var1 “is less than” $var2
elif [ $var1 –gt $var2 ]
then
        echo $var1 “is greater than” $var2
else
        echo $var1 “is equal to” $var2
fi
```

*

Testing Strings

- Performing string comparisons. It is a good idea to put the shell variable being tested inside double quotes.

[“$1” = “yes”]

[“$2” != “no”]

- can use = and != with STRINGS , not numbers

- Note that the following will give a syntax error when $1 is empty since:

[$1 != “no”]

- becomes

[!= “no”]

More on String Relational Operators

- The set of string relational operators are:

{ =, !=, >, >=, <, <= }

- The { >, >=, <, <= } operators assume an ASCII ordering (for example “a” < “c”). **These operators are used with the expr command that computes an expression. The backslash has to be used before the operators so that they are not confused with I/O redirection**

Testing with Multiple Conditions

- && is the *and* operator
- || is the *or* operator
- checking for the *and* of several conditions

[“$1” = “yes”] && [-r $2.txt]

[“$1” = “no”] && [$# -eq 1]

- checking for the *or* of several conditions

[“$1” = “no”] || [“$2” = “maybe”]

Quoting Rules

- Using single quotes

‘xyz’ disables all special characters in xyz

- Using double quotes

“xyz” disables all special characters in xyz except $, `, and \.

- using the backslash

\x disables the special meaning of character x

*

Quoting Examples

```
var1="alpha"        #set the variable
echo $var1          #prints: alpha
echo "$var1"        #prints: alpha
echo '$var1'        #prints: $var1
cost=2000
echo 'cost:$cost'       #prints: cost:$cost
echo "cost:$cost"       #prints: cost:2000
echo "cost:\$cost"      #prints: cost:$cost
echo "cost:\$$cost"     #prints: cost:$2000
```

*

Using Exit

- The exit command causes the current shell script to terminate. There is an implicit exit at the end of each shell script. The exit command can set the *status* at the time of exit. If the status is not provided, the script will exit with the status of the last command.
- General form:

exit

- or

exit <status>

- $? is set to the value of the last executed command
- Zero normally indicates success. Nonzero values indicate some type of failure. Thus, exit 0 is normally used to indicate that the script terminated without errors.

Exit Command (again)

- Conventionally, zero normally indicates success. Nonzero values indicate some type of failure. It is thus good practice to ensure that if the shell script terminates properly, it is with an “exit 0” command.
- If the shell script terminates with some error that would be useful to a calling program, terminate with an “exit 1” or other nonzero condition.
- most Unix utilities that are written in C will also call “exit(<value>);” upon termination to pass a value back to the shell or utility that called that utility.

Exit Example

- The following shell script exits properly. It also distinguishes the response through the value returned.

```
#!/bin/sh
#determines a yes (0) or no (1) answer from user
echo “Please answer yes or no”; read answer
while :
do
  case $answer in
  “yes”) exit 0;;
  “no”) exit 1;;
  *) echo “Invalid; enter yes or no only”
        read answer;;
  esac
done
```

Testing the Exit Status

- Conditions tested in control statements can also be the exit status of commands. Assume that the script "yes.sh" has been invoked.
- The following segment will test this as part of its script:

```
if yes.sh
then
  echo "enter file name"
  read file
else
  echo "goodbye"; exit 0
fi
```

*

Case Statement (strings only)

- Compares stringvalue to each of the strings in the patterns. At a match, it does the corresponding commands. ;; (similar to "break" in c++) indicates to jump to the statement after the esac (end of case). *) means the default case.
- Form:

```
case stringvalue in
pattern1) one or more commands;;
pattern2) one or more commands;;
…
          *) one or more commands;;
esac
```

*

Case Statement Example

```
echo “do you want to remove  file $1?”
echo “ please enter yes or no”
read ans
case $ans in
“yes”) rm $1
        echo “file removed”
        ;;
“no”) echo “file not removed”
        ;;
    *) echo “do not understand your request”
esac
```

*

while and until statements

- while form:

while <condition>
do
    one or more commands
done

- until form:

until <condition>
do
    one or more commands
done

*

while and until examples

```
read cmd
while [ $cmd != “quit” ]
do
        …
        read cmd
done
######### note, both examples achieve the same result.
read cmd
until [ $cmd = “quit” ]
do
        …
        read cmd
done
```

*

For statement (loop)

- The shell <variable> is assigned each word in the list, where the set of commands is performed each time the word is assigned to the variable. If the "in <word_list>" is omitted, then the variable is assigned each of the command line arguments.

for <variable> [in <word_list>]

do

one or more commands

done

*

Using for in a directory (USEFUL!)

- Use the for loop to iterate through every file in a directory

```
for filename in *
do
        echo $filename
done
```

- You can replace * with *.doc to iterate through only .doc files in the current directory.
- To look inside a directory other than CWD, do:

for filename in <relativepathname>/*

ex: for filename in myDir/*

For statement examples

```
#!/bin/sh
#makes a backup of certain files and echoes arguments
for file in `ls *.c`
do
  cp $file $file.bak
done
for arg
do
echo $arg
done
exit
```

*

Command Substitution

- a string in back quotes ` … ` does command substitution
  - This means that the result of the command (the standard output of the command) replaces the back quoted string

Examples:

```
count=`wc -w <$1`
```

the value of count is assigned the number of words in file $1

```
if [ `wc -l < $2.txt` -lt 1000 ];
```

#checks if the number of lines in the file is < 1000

```
cat `grep -l exit *.sh`
```

#print out all *.sh files containing the word exit

*

Expr

- expr evaluates an arithmetic or relational expression and prints its result to standard output. This is useful when you need to perform calculations in the shell script. It outputs 1 (true) or 0 (false) when evaluating a relational expression.
- Note that the arguments are operators must be separated by spaces.
- Example from the tcsh command line (note *set*)

```
set alpha = 3
expr $alpha + 2 #result printed out is 5
```

*

More expr examples

```
var=`expr $var + 1`            #increment var by 1

if [ `expr $s1 \< $s2` = 1 ]   #check if the value of  s1
                               #is less than value of s2

beta=`expr $beta \* 2`         #multiply value of beta by 2
set beta = 10; expr $beta / 2  #using tcsh directly, result is 5

expr "$alpha" = hello          #output 1 if variable alpha is
                               #hello
```

*

Good reference on scripting

- http://steve-parker.org/sh/sh.shtml

Also, see examples on blackboard.

*