# Bash Scripting: Advanced Topics

CISC3130, Spring 2013

Dr. Zhang

# Outline

- Review HW1, HW2 and Quiz2
- Review of standard input/output/error
  - How to redirect them ?
  - Pipeline
- Review of bash scripting
- Functions
- Here documents
- Arrays

# Homework 2

- **match phone numbers in text file**
  - 7188174484, 718-817-4484, (718)817,4484
  - 817-4484, or 817,4484, 8174484.
  - (01)718,817,4484, 01,718-817-4484
- **grep -f phone.grep file.txt** , where phone.grep:

```
[^0-9][0-9]\{10\}$
[^0-9][0-9]\{10\}[^0-9]
[^0-9][0-9]\{3\}\-[0-9]\{3\}\-[0-9]\{4\}$
[^0-9][0-9]\{3\}\-[0-9]\{3\}\-[0-9]\{4\}[^0-9]
[^0-9][0-9]\{3\}\,[0-9]\{4\}$
[^0-9][0-9]\{3\}\,[0-9]\{4\}[^0-9]
[^0-9][0-9]\{3\}\-[0-9]\{4\}$
[^0-9][0-9]\{3\}\-[0-9]\{4\}[^0-9]
```

Match 10 digits at end of line

Match 10 digits, and a non-digit char

718-817,4484 at end of line

# Homework 2

[^0-9]*\([0-9]\{2\}\)\([0-9]\{3\}\)[0-9]\{3\}\,[0-9]\{4\}$

[^0-9]*\([0-9]\{2\}\)\([0-9]\{3\}\)[0-9]\{3\}\,[0-9]\{4\}[^0-9]

[^0-9]*\([0-9]\{2\}\)\([0-9]\{3\}\))?[0-9]\{3\}\-[0-9]\{4\}$

[^0-9]*\([0-9]\{2\}\)\([0-9]\{3\}\))?[0-9]\{3\}\-[0-9]\{4\}[^0-9]

[^0-9]*[0-9]\{2\}\,[0-9]\{3\}\,[0-9]\{3\}\,[0-9]\{4\}$

[^0-9]*[0-9]\{2\}\,[0-9]\{3\}\-[0-9]\{3\}\-[0-9]\{4\}[^0-9]

[^0-9]*[0-9]\{2\}\,[0-9]\{3\}\-[0-9]\{3\}\-[0-9]\{4\}$

[^0-9]*[0-9]\{2\}\,[0-9]\{3\}\,[0-9]\{3\}\,[0-9]\{4\}[^0-9]

# Homework 2

- Write a sed script file that remove all one-line comments from C/C++ source code files. Note that such comments starting with //, and ends at the end of line. You need to take care the cases where // appears in double quote, or single quote, in thsse cases, what comes after // is not comment.
  - **rmcnt.sed** :

```
#!/bin/sed -f
## remove one-line comments from C/C++ code
/^[^"']*\/\// s/\/\/.*$/ /g
```

Apply to lines that contain // not preceding by ' or "

Replace // and following chars with space

  - rmcnt.sed sample.cpp

# Quiz 2

- How to write to standard output in shell script:

```
#!/bin/bash
echo "Hello world";
echo "Something is wrong" 1>& 2
ls ABCDEF 2>&1
```

- Try it out:
  - ./test_redirect.sh 2> err_out  ## what happens?
  - ./test_redirect.sh > std_out  ## what happens?
  - ./test_redirect.sh > std_out 2>&1

# Quiz2

- Mark constants with < and >

```
#/bin/bash
# First find numerica constants in the code
#grep -E '[^a-zA-Z_][0-9]+\.?[0-9]+' $1

# now mark constants with <>
echo mark constants in file $1
sed 's/\([^a-zA-Z0-9_]\)\([0-9][0-9]*\.\{0,1\}[0-9][0-9]*\)/\1\<\2\>/g' $1
```

The char before constant:
not alphabet, not _, and not digit

A numeric constant:
Optional decimal points: \.\{0,1\}
cannot use ?, as sed use BRE

# Standard input/output/error

- By default, link to keyboard and terminal window respectively
  - Can be redirected to files
  - Can be redirected to pipeline
    - input can be redirected to reading end of a pipe
    - output and error can be redirected to writing end of a pipe
- When a bash script's input/output/error is redirected:
  - E.g., headtail 3 10 .bash_profile > output
  - ls –l | headtail 10 24 | wc –l
  - input/output/error for every command in the script are redirected !

# Save standard input if necessary

```
#!/bin/bash
# Count # of lines, and search for phone in a file; if a file is
# not specified, process standard input
set -x          ## turn on execution tracing
if [ $# -eq 0 ]
then
     cat > stdinput  ## save standard input to a file
    set stdinput
fi
## so that we can use as many times as we want
wc –l $*
grep  -f phone.grep $*
rm stdinput
exit 0
```

Or use –x in
first line, i.e., #!/bin/bash –x

Or type
$ bash -x countlines_searchphoneno.sh
to run the scripts

Code at:
countlines_searchphoneno.sh

# Redirection can be applied to loop

```
for i in `ls *.sh`
do
    echo $i
    cat $i
done > all_shellscripts
```

```
rm all_shellscripts
for i in `ls *.sh`
do
    echo $i >>all_shellscripts
    cat $i >>all_shellscripts
done
```

Similar for <, |

# case construct: branching

- **case** construct is analogus to *switch* in C/C++.

```
case "$variable" in
 shellpattern1 )
 command…
 ;;
 shellpattern2)
 command …
 ;;
 shell pattern n)
 command…
 ;;
esac
```

- Quoting variables is not mandatory
- Each pattern can contain **shell wildcard** (*,?,[a-z]), ends with a **)**
- Each condition block ends with **;;**
- If a condition tests *true*, then associated commands execute and the **case** block terminates.
- entire **case** block ends with an **esac**

# Calculator using case block

```
case  "$op" in
"+" )       result=$(($x + $y))
                echo $x $op $y = $result;;
"-" )       result=$(($x - $y))
                echo $x $op $y = $result;;
"*" )       result=$(($x * $y))
            echo $x \* $y = $result;;
"/" )       result=$(($x / $y))
            echo $x $op $y = $result;;
* )       echo Unknow operator $op;;
esac
```

```
#!/bin/bash
OPT=$1 # option
FILE=$2 # filename
 # test -e and -E command line args matching
case $OPT in
 -e|-E)
    echo "Editing $2 file..."
    # make sure filename is passed else an error displayed
    [ -z $FILE ] && { echo "File name missing"; exit 1; } || vi $FILE ;;
-c|-C)
    echo "Displaying $2 file..."
     [ -z $FILE ] && { echo "File name missing"; exit 1; } || cat $FILE ;;
 -d|-D)
    echo "Today is $(date)" ;;
*)
    echo "Bad argument!"
    echo "Usage: $0 -ecd filename"
    echo " -e file : Edit file."
    echo " -c file : Display file."
    echo " -d  : Display current date and time." ;;
esac
```

Lazy evaluation of && and ||

test if string is null

# Case example

```
case $1 in
-f)
    ## case for –f option
    ;;
-d | --directory)
   ## -f or –directory option
    ;;
*)
   echo $1: unknown option >&2
   exit 1;
esac
```

# More about bash loop structures

# Infinite loop

```
while [ 1 ]
do
        echo -n "Enter your password"
        read input
        if [ $input = "secret" ]
        then
                break  ## break out of the loop
        else
                echo -n "Try again… "
        fi
done
```

# continue command

- Continue from the top of the for loop
  - Ignore rest of commands in the loop, and continue the loop from the top again (for the next value in the list)

```
i=1
for day in Mon Tue Wed Thu Fri Sat Sun
do
    echo -n "Day $((i++)) : $day"
    if [ $i -eq 7 -o $i -eq 8 ];
    then
      echo " (WEEKEND)"
      continue;
    fi
    echo " (weekday)"
done
```

# For loop without a list

```
#!/bin/bash

for i
do
  echo hello $i
done
```

# For loop

```
i=1
for username in `awk -F: '{print $1}' /etc/passwd`
do
    echo "Username $((i++)) : $username"
done
```

# Loop through files/directories

- loop through files and directories under a specific directory

```
i=1
cd ~
for item in *
do
    echo "Item $((i++)) : $item"
done
```

# C-Style for loop

```
for (( EXP1; EXP2; EXP3 ))
do
    Command1
    …
    Commandn
done
```

EXP1: initializer
EXP2: a loop-test or condition
EXP3: counting expression

- *Example:*

```
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
  echo "Welcome $c times"
done
```

# Select loop

- **select** construct: allows easy menu generation

```
select WORD [in LIST]
do
   RESPECTIVE-COMMANDS;
done
```

1. List of items printed to standard error, each item preceded by a number.
   - If **in LIST** is not present, positional parameters (command line arguments) are used
2. A prompt is printed, one line from standard input is read.
   1. If input is a number corresponding to one of items, value of WORD is set to name of that item.
   2. If line is empty, items and the PS3 prompt are displayed again.
   3. If an *EOF* (End Of File) is read, loop exits.
3. **RESPECTIVE-COMMANDS** are executed after each selection
4. Go back to 1

# select construct: example

```
#!/bin/bash
OPTIONS="Hello Quit“
 select opt in $OPTIONS; do
      if [ "$opt" = "Quit" ]
      then
          echo done
          exit
      elif [ "$opt" = "Hello" ]
      then
           echo Hello World
      else
           echo bad option
      fi
done
```

~zhang/public_html/cs3130/Codes/select_ex

# Next:

- More advanced bash scripting
  - Array
  - Function
  - Inline input, or here document

# Array

- Bash provides one-dimensional array variables
- Assign values to array:

```
array=( one two three )
files=( "/etc/passwd" "/etc/group" "/etc/hosts" )
limits=( 10  20  26  39  48)
```

- Access array element : ${array_name[index]}
  - indexed using integers and are zero-based.

  ${array[1]}
- To access all items in arary: ${array_name[*]} , ${array_name[@]}
- To access array length: len=${#x[@]}

# To Iterate Through Array Values

```
#!/bin/bash
# declare an array called array and define 3 vales
 array=( one two three )
for i in "${array[@]}"
do
    echo $i
done
```

# Exercise/Example

- Write a script that read a sequence of numbers and save them in an array, print out the array content and size.
- Usage: EchoNumber [file]
  - If no file is specified, read from standard input

- Example script:
  - LargestSmallest.sh

```
#!/bin/bash
i=0
if [ $# -eq 0 ]
then
        echo "Enter the numbers, Ctrl-D to end";
        cat > stdinput
        set stdinput
fi

while read num
do
        a[$i]=$num
        i=$((i+1))
done < $1

echo Array is ${a[*]}, with ${#a[*]} numbers
```

# Bash function

- Functions: to increase modularity and readability
  - More efficient than breaking scripts into many smaller ones
- Syntax to define a function:

```
function functionname()
{
    commands . .
}
```

  - **function** is a keyword which is optional.
  - **functionname** is the name of the function
    - No need to specify argument in ( )
  - **commands** – List of commands to be executed I
    - **exit status** of the function is exit status of last command executed in the function body.

# Function call

- Call bash function from command line or script
  - $ functionname arg1 arg2
  - When shell interprets a command line, it first looks into the special built-in functions like break, continue, eval, exec etc., then it looks for shell functions.
- function defined in a shell start up file (e.g.,.bash_profile ).
  - available for you from command line every time you log on

# About functions

- **Parameter passing**: $1, $2, …
- **Result returning**
  - Use echo command
  - Through setting a variable
  - return command: to return an exit status

```
#! /bin/bash
calsum(){
        echo `expr $1 + $2`
}
x=1;y=2;
sum=`calsum $x $y`
```

```
calsum(){
        sum=`expr $1 + $2`
}
x=1;y=2;
calsum $x $y
echo z=$sum
```

# About functions

- Local variable: its scope is within the function

```
#! /bin/bash
calsumsqr(){
        local sum=`expr $1 + $2`;
        echo `expr $sum * $sum`
}
x=1;y=2;
z=`calsum $x $y`
```

# Exercise/Example

- Write a function that check whether a user is log on or not (CheckUser.sh)

```
function UserOnline()
{
   if who | grep $1          ## UserOnline takes a parameter
   then
      return 0        ## 0 indicates success
   else
      return 1       ##1 for failure, i.e., offline
  fi
}
if UserOnline $1                ## function's return value as condition/test
then
      echo User $1 is online
else
      echo User $1 is offline
fi
```

# Here document (inline document)

- A special way to pass standard input to a command: here document, i.e., from shell script itself
- Benefits: store codes and data together, easier to maintain

```
#!/bin/bash
cat <<!FUNKY!
Hello
This is a here
Document
!FUNKY!
```

Here document starts with <<, followed by a special string which is repeated at the end of the document.

Note: the special string should be chosen to be rare one.

# Here document:2

- Example: 411 script that looks up a phone book
  - Usage example: 411 joke

```
#!/bin/bash
 grep "$*" << End
 Dial-a-joke 212-976-3838
 Dial-a-prayer 212-246-4200
 Dial santa 212-976-141
 End
```

# Phone # searching script

```
#!/bin/bash
cat > phone.pattern << PATTERNS
[^0-9][0-9]\{10\}$
[^0-9][0-9]\{10\}[^0-9]
[^0-9][0-9]\{3\}\-[0-9]\{3\}\-[0-9]\{4\}$
[^0-9][0-9]\{3\}\-[0-9]\{3\}\-[0-9]\{4\}[^0-9]
[^0-9][0-9]\{3\}\,[0-9]\{4\}$
PATTERNS

grep -f phone.pattern $*
rm phone.pattern          ##no need to keep this file …
```

# A case study: bundle program

- Suppose a friend asks for copies of shell files in your bin directory

```
$  cd ~/bin
$  for i in *.sh
> do
>      echo ===== This is file $i ==============
>      cat $i
>  done | mail yourfriend@hotmail.com
```

Pipeline & input/output redirection can be applied to for, while, until loop.

# Make it better ?

- Construct a mail message that could automatically unpack itself, i.e., to generate the original files packed inside
  - E.g., A shell script contains instructions for unpacking, and the files content themselves
  - Use here document mechanism

# A bash script contains two files

```
#To unbundle, bash this file
echo file1 1>&2
cat >file1 <<'End of file1'
A
B
C
End of file1
echo  file2 1>&2
cat >file2 <<'End of file2'
1
2
3
end of file2
```

What does this script do?

How to create such bundle file automatically?
Use a script, bundle.sh file1 file2 file3 …

# Bundle script

```
#!/bin/bash
## write a shell script that contains files specified in arguments
echo '#!/bin/bash'
echo '# To unbundle, bash this file'
for i                    ## without a list of items, loop through
                         ## command line arguments
do
        echo "echo $i 1>&2"
        echo "cat >$i <<'End of $i'"
        cat $i
        echo "End of $i"
done
```



For homework 3, use bundle.sh to generate a hw3_bundle file, and submit it.

# Summary

- Review of shell scripting
- Examples
- Array, function, inline document

# Outline

- Coding standard: how to get good grades in lab assignment
- Review of standard input/output/error
  - How to redirect them ?
  - Pipeline
- Review of bash scripting

# Bash scripting: general hints

- Use echo command to trace (like cout, printf in C/C++)
- Sometimes there are alternatives ways to do things, choose one and remember it:
  - $(( …)), and $[ … ]
  - [[ ]] for test
- Be careful about typo, shell wont complain variable not declared/assigned …
  - The price of freedom
- A walk-through of basic bash scripting

# Bash Scripting

- Variables
  - Environment variable: affect behavior of shell
  - User defined variable: default type is string, can declare it to be other type
  - Positional variables: used to pass command line arguments to bash script
- Variable assignment:

  x=10  ## assign value 10 to variable x, no space around =

  x=$x+1  ## add 1 to x's value and assign to x

  PATH=$PATH:.:~/bin
- To refer to a variable's value,  precede variable name with $

# A script that display positional variable

```
echo All arguments: $*
echo Number of arguments: $#
echo Script name: $0
echo argument 1: $1
echo argument 2: $2

for arg in $*
do
   echo argument $arg
done
```

# arithmetic operation

- As variable's default type is string, to perform arithmetic operation, use the following syntax

  $[$x+1] or $(($x+1))

- For simpler syntax: declare variable to be numerical

  declare –i x

  x=$x*10+2

- Above are for integer arithmetic operations only ..

# Command bc

- An arbitrary precision calculator

```
$ bc
3.14159*10^2
314.15900
130^2
16900
sqrt(1000)
31
scale=4
sqrt(1000)
31.6277
quit
```

An interactive calculator:
 * user input shown in normal font,
 * result shown in italics font

Internal variable **scale:**
  * control the number of decimal points after decimal point

# bc in command line/script

- To evaluate an expression, simply send it using pipe to bc

  echo "56.8 + 77.7" | bc

- Write a script that read a Fahrenheit degree from standard input, convert it to Celsius degree (up to 2 digits after decimal point):

  C=(F-32)*5/9

- Base conversion, from base 10 (decimal) to base 16 (hexadecimal)

  echo "obase=16; ibase=10; 56" | bc

# Test/Conditions

- Any command or script, if it return 0, then test is successful

```
if  rm tmp.txt
then
    echo file tmp.txt has been deleted
else
    echo fail to remove file tmp.txt
Fi
```

- Use !  to negate

# Test Condition

- Numerical comparisons
  - -lt, -ne, -ge, …
- String comparision
- Pattern matching: using double brackets
  - To test if first argument is "–" followed by a number:

```
if [[ "$1" ==  -[0-9]* ]]
then
    ….
```