# Lecture Notes for Lecture 4 of CS 5600 (Computer Systems) for the Fall, 2019 session at the Northeastern University Silicon Valley Campus.

## *Shell Scripting*

Philip Gust,
Clinical Instructor
Department of Computer Science

# Lecture 3 Review

- In lecture 3, we learned about *shells* that run in the outer-most layer of the operating system.
- Shells are interactive command interpreters that provide access to functionality provided by the libraries and system calls. They also provide functions to facilitate user interaction.
- Shells provide access to commands that implement higher-level functions, and also manage the execution of applications and services on behalf of users.
- We also looked at some basic POSIX commands, and ways that they can perform useful tasks.

# Shell Scripting

- In this lecture, we will learn about shells as an environment for creating script applications.
- Shell scripts utilize the same set of commands and techniques for composing them that we learned about in the previous lecture.
- When creating shell scripts, these commands are used in the same way that library functions are used in regular programming languages.
- We will now learn about the facilities available in the bash shell language for creating full-functioned script applications, including control structures, variables, functions and I/O operations.
- We will also see how to combine script-based and conventional utilities to create more complex applications.

# Shell Scripting

**Create a Script File**

- The shell is also a scripting language can include built-in and external commands, variables, conditional and loop control structures, and functions.
- Any sequence of commands that can be typed into the shell can also be included in a shell script and performed repeatedly by running the script from the shell.
- In POSIX systems, scripts and compiled programs are treated equally.  A script can execute a command written in C, or a shell script.

# Shell Scripting

## Create a Script File

- Here is a simple shell script that reports the number of arguments that you passed to the script on the command line. We will store it in a file "hello.sh" (the suffix is not required).

```
#!/bin/bash
# file name: "hello.sh"
echo Hello, you passed  $# parameters
```

- Parameters are accessed by position: $1 for the first parameter, $2 for the second one, etc. $# is the number of parameters.
- You can run the script in one of two ways:
  - Run **bash** and pass in the script and parameters
  - Make the script "executable" to run as a normal program:
    **chmod** +x hello.sh

# Shell Scripting

**Create a Script File**

- Passed in to *bash* with parameters

  **bash** ./hello.sh

  **bash** ./hello.sh world

  **bash** ./hello.sh from around the world

- Run as executable

  ./hello.sh

  ./hello.sh world

  ./hello.sh from around the world

- Output is

  Hello, you passed  0 parameters

  Hello, you passed 1 parameters

  Hello, you passed 4 parameters

# Shell Scripting

## Create a Script File

- Parameters can be accessed sequentially using the shift command.
- The shift command shifts the parameter list left *n* positions. The default value of *n* is 1.
- After shifting 1, the next parameter is now at $1, the one after that at $2, etc., and the value $# is decreased.

| command | parameter list | value of $1 | value of $# |
|---|---|---|---|
| | from around the world | from | 4 |
| **shift** | around the world | around | 3 |
| **shift** | the world | the | 2 |
| **shift** | world | world | 1 |
| **shift** | | | 0 |

# Shell Scripting

**Variables**

- You can store values for later use using *shell variables*.  These are also referred to as *environment variables*. For example:
  - proj_path="~/my_project"
    Defines the variable "proj_path" with the path "~/my_project"
- By prefixing a dollar sign, "$"  to the variable name or enclosing variable name in ${}, you can access its value:
  - **echo** $proj_path
    Prints "~/my_project" if "proj_path" is defined as above
  - **cd** ${proj_path}/bin
    Changes working directory to "~/my_project/bin" if "proj_path" is defined as above
  - proj_bin=${proj_path}/bin
    Defines variable "proj_bin" as "~/my_project/bin" if "proj_path" is defined as above

# Shell Scripting

**Variables**

- Unset a variable using the "unset" command:
  - **unset** proj_path
    variable no longer defined
- List all currently defined variables using the "env" command. The shell pre-defines and maintains a number of variables.

```
$ env
HOSTNAME=phil-local
SHELL=/bin/bash
HISTSIZE=100
USER=phil
LOGNAME=phil
HOME=/Users/phil
PWD=/Users/phil
PATH=/bin:/usr/bin:/usr/local/bin:/usr/sbin:/Users/phil/bin
TERM=xterm
TEMP=/tmp
LANG=en_US.UTF-8
```

# Shell Scripting

**Quoting and Variable Substitution**

- Bash supports both single and double quotes. Quotes are used to group together multiple words separated by whitespace.
- Variable substitution takes place within double quotes, but no substitution takes place within single quotes.

```
str=hello                    #quotes not required
str="good morning"           #quotes required
echo "$str everyone"         #substitution within double quotes
echo '$str everyone'         #no substitution within single quotes
echo "she said \"$str\""     #escaped quotes within quotes
```

Output:

```
good morning everyone
$str everyone
she said "good morning"
```

# Shell Scripting

**Capture to Variables**

- The output of a command can be captured and either used directly or assigned to a variable by enclosing the command in $()
    - **wc** –c  $(**find** . –type f –name '*.txt')
      count number of words in all text files in the current directory tree
    - dirlist=$(**ls**)
      capture list of files in the current directory
    - curdate=$(**date** +%Y-%m-%d)
      capture current date as "2019-01-30"

# Shells and System Commands

**Reading Into Variables**

- A line of input can be read into one or more variables and used with other commends:
  - **read** -r line
    read the entire line into  variable line, trims leading/trailing ws.
  - IFS= **read** -r line   # IFS set just for this command
    read the entire line into  variable line preserving leading/trailing ws.
  - **read** -r var1 var2 var3
    reads words delimited by whitespaces from input into three variables; the last variable gets the remainder of the line.
  - IFS=':' **read** -r var1 var2 var3  <<< "word1:word2:word3"
    reads words delimited by colons from input into three variables.

    IFS stands for Input Field Specifier. The default value of the IFS variable is  whitespace characters (<space>, <tab>, and <newline> ).

# Shells and System Commands

**Formatted Output**

- One or more variables can be formatted using the **printf** command. The command has a formatting string with field specifiers, and zero or more values to be formatted.
  - **printf** "My name is %s\n" “$(**whoami**)”
    Prints the user's login name from the **whoami** command (note could use shell supplied $USER variable)
  - **printf** "Today is %s\n" $(**date** +%Y-%m-%d)
    Prints today's date captured from the **date** command
  - **printf** "Current directory %s has %d entries\n" $(**pwd**) $(**ls** | **wc** -l)
    Prints current working directory and number of files it contains from **pwd**, **ls**, and **wc** commands. (could use shell supplied $PWD variable)
  - **printf** "%d\n" 1 2 3 4 5
    Prints the numbers 1, 2, 3, 4, 5 on separate lines since there are more values than format specifiers

# Shell Scripting

**Numeric Variables**

- Bash supports integer arithmetic expressions by enclosing the expression in $(()). Supports operators +, -, *, /, ++, --, %, ==, !=, <, >, >=, <=, <<, >>, &, |, ^, ~
  - declare –i  m=5    declare integer variable (add –r for read-only)
  - m="abc"    invalid: sets integer-only variable m to 0
  - n=5    initialize general variable n to an integer
  - $(( n+1 ))    evaluates to 6
  - $(( n % 2 ))    evaluates to 1
  - $(( n*n + 2*n + 1 ))    evaluates to 36
  - $(( n++ ))    evaluates to 5 and increments n to 6
  - $(( --n ))    decrements n to 5 and evaluates to 5
  - $((n == 5))    evaluates to 1 (true)
  - $((n != 5))    evaluates to 0 (false)

# Shell Scripting

**String Variables**

- Bash supports operations on string variables
  - s="aaaa cccc eeee"
    initialize s to the string "aaaa cccc eeee"
  - ${#s}
    evaluates to the length of the string: 14
  - ${s:5:4}
    evaluates to "cccc", the four characters at positions 5, 6, 7, and 8
  - ${s:10}
    evaluates to "eeee", the substring starting at position 10
  - ${s:0:5}CCCC${s:9}
    evaluates to string replacing "cccc" with "CCCC": "aaaa CCCC eeee"
  - The numeric values can also be numeric variables.

    ```
    i=10
    ${s:i}   # evaluates to "eeee"
    ```

# Shell Scripting

**String Variables**

- Bash supports operations on string variables
  - s="aaaa cccc eeee":
    initialize s to the string "aaaa cccc eeee"
  - ${s/a/A}
    replaces the first 'a' with 'A' ("Aaaa cccc eeee")
  - ${s//[aeiou]/?}
    replaces all vowels with question marks ("???? cccc ????")
  - ${s#a*a}
    deletes shortest match of "a*a" from front ("aa cccc eeee")
  - ${s##a*a}
    deletes longest match of "a*a from front (" cccc eeee")
  - ${s%e*e}
    deletes shortest match of "e*e" from back ("aaaa cccc ee")
  - ${s%%e*e}
    deletes longest match of "e*e" from back ("aaaa cccc ")

# Shell Scripting

## Array Variables

- Bash supports array variables. Here are examples.

| | |
|---|---|
| declare –a arr | Create an empty array (optional, add –r for read-only) |
| arr=() | Create an empty array |
| arr=(1 2 3) | Initialize array |
| readarray –t arr | Reads lines of stdin into array; -t strips newlines (bash 4) |
| ${arr[2]} | Retrieve third element |
| ${arr[@]} | Retrieve all elements |
| ${!arr[@]} | Retrieve array indices |
| ${#arr[@]} | Calculate array size |
| arr[2]=3 | Overwrite element 2 |
| unset arr[2] | Unset element 2 |
| arr+=(4) | Append value(s) |
| arr=( $(ls) ) | Save ls output as an array of files |
| ${arr[@]:s:n} | Retrieve n elements starting at index s |

# Shell Scripting

## Associative Array Variables

- Bash 4 and later supports associative array variables. Here are examples.

| | |
|---|---|
| **declare** -A arr | Create associative array (add –r for read-only) |
| arr=([mon]=1 [tue]=2 [wed]=3) | Initialize associative array of weekdays |
| ${arr[wed]} | Retrieve "wed" element (must user ${ } ) |
| ${arr[@]} | Retrieve all values |
| ${!arr[@]} | Retrieve keys |
| ${#arr[@]} | Calculate array size |
| arr[wed]=9 | Overwrite element "wed" |
| **unset** arr[wed] | Unset element "wed" |

# Shell Scripting

**Variable Indirect and Nameref (Alias)**

- You can access the value of a variable indirectly through another variable that contains its name (like C/C++ pointer vars).
  - proj_path="~/my_project"
    path_name="proj_path"
    echo ${!path_name}

    echoes "~/my_project"
- You can create an *nameref* (alias) for another variable and use it wherever the other variable can be used (like C++ reference vars).
  - arr=(1 2 3)
    declare –n  arr_alias="arr"
    arr_alias[1]=-2
    echo ${arr[1]} ${arr_alias[1]}

    echoes -2 -2

# Shell Scripting

**Test**

- The built-in bash **test** command can test file attributes, and perform string and arithmetic comparisons. Test has no output, but returns exit status 0 for “true” and 1 for “false”
- Example:

```
num=4
test $num –gt 5
echo $?    # echo return status of last command
num=6
test $num –gt 5
echo $?    # echo return status of last command
```

- Output:

```
1
0
```

# Shell Scripting

**Test**

- Bash provides another way to invoke the built-in **test** command by enclosing the expression in square brackets: **[** *expr* **]**
- Example:

```
num=4
[ $num –gt 5 ]
echo $?   # echo return status of last command
num=6
[ $num –gt 5 ]
echo $?   # echo return status of last command
```

- Output:

```
1
0
```

# Shell Scripting

## Numeric Tests

| Operator | Description |
|---|---|
| **INTEGER1 -eq INTEGER2** | INTEGER1 is numerically equal to INTEGER2 |
| **INTEGER1 -ne INTEGER2** | INTEGER1 is numerically not equal to INTEGER2 |
| **INTEGER1 -lt INTEGER2** | INTEGER1 is numerically less than INTEGER2 |
| **INTEGER1 -le INTEGER2** | INTEGER1 is numerically less or equal to INTEGER2 |
| **INTEGER1 -gt INTEGER2** | INTEGER1 is numerically greater than INTEGER2 |
| **INTEGER1 -ge INTEGER2** | INTEGER1 is numerically greater or equal INTEGER2 |

# Shell Scripting

## String Tests

| Operator | Description |
|---|---|
| **! EXPRESSION** | The EXPRESSION is false. |
| **-n STRING** | The length of STRING is greater than zero. |
| **-z STRING** | The length of STRING is zero (ie it is empty). |
| **STRING1 = STRING2** | STRING1 is equal to STRING2 |
| **STRING1 != STRING2** | STRING1 is not equal to STRING2 |

# Shell Scripting

## File Tests

| Operator | Description |
|---|---|
| **-d FILE** | FILE exists and is a directory. |
| **-e FILE** | FILE exists. |
| **-r FILE** | FILE exists and the read permission is granted. |
| **-s FILE** | FILE exists and it's size is greater than zero (ie. it is not empty). |
| **-w FILE** | FILE exists and the write permission is granted. |
| **-x FILE** | FILE exists and the execute permission is granted. |

# Shell Scripting

**Alternate Numeric Test**

- Bash provides another way to test numeric expressions using the double-parentheses numeric evaluation notation we saw earlier.
- Example:

```
num=4
((num > 5))
echo $?   # echo return status of last command
num=6
((num > 5))
echo $?   # echo return status of last command
```

- Output:

```
1
0
```

# Shell Scripting

**Compound Tests**

- Compound tests can be connected using '||' and '&&'.

```
value=50
[  $value –gt 100 ] || [ $value –lt 10 ]
echo $?
((value <= 100)) && ((value >= 10))
echo $?
fi
```

- Outputs

```
1
0
```

# Shell Scripting

## *If* Statement

- A basic **if** statement executes commands if one or more conditions is true, or optional **else** commands

  **if** *<test1>*

  **then**

      <commands>

  **elif** *<test2>*

  **then**

      <commands>

  **else**

      <commands>

  **fi**

# Shell Scripting

## *If* Statement

- This example tests the a numeric value against three ranges.

```
value=1000
if [  $value –gt 100 ]        # or (( value > 100 ))
then
    echo Hey, $value is too large
elif [ $value –lt 10 ]        #or (( value < 10 ))
then
    echo Hey, $value is too small
else  # 10 < value <= 100
    echo Hey, $value is too just right
fi
```

- Outputs "Hey, 1000 is too large"

# Shell Scripting

## **C*ase* Statement**

- Case statement selects commands to execute based on selector value. Targets can be literals or file patterns, including wildcards and character or digit ranges.

**case** *<selector value>* **in**<br>
*target1*)<br>
    *<commands>*<br>
    *;;*<br>
*target2*)<br>
    *<commands>*<br>
    *;;*<br>
*target3*)<br>
    *<commands>*<br>
    *;;*<br>
*)<br>
    *<commands>*<br>
    *;;*<br>
**esac**

# Shell Scripting

## *Case* Statement

- This example matches the number with a digit range.

```
case "$val" in
    [1-3]) echo low
    ;;
    [4-6]) echo medium
    ;;
    [7-9]) echo high
    ;;
    *) echo try again
    ;;
esac
```

# Shell Scripting

### ***While* Loop**

- While loops repeat commands while an expression is true. They have the following format:

  **while** *<some test>*

  **do**

      *<commands>*

  **done**

- Anything between **do** and **done** will be executed while the test (between the square brackets) is true.
- Use **break** to break out of loop, **continue** for next iteration.

# Shell Scripting

***While* Loop**

```
counter=1
while [ $counter -le 10 ]    # or (( counter <= 10 ))
do
    echo $counter
    ((counter++))
done
echo All done
```

- Outputs the number 1 through 10, and then "All done"

# Shell Scripting

## ***While*** **Loop**

```
# initialize an associative array by breaking apart lines
declare -A arr        # bash 4+
while IFS='=' read -r name val   # use = as word separator
do
   arr[$name]="${val}"
done  << EOF
mon=Monday
tue=Tuesday
wed=Wednesday
EOF
printf "Day for 'mon' is '%s'\n" "${arr[mon]}"
```

- Outputs  "Day for 'mon' is 'Monday'"

# Shell Scripting

## Combining *while* and *case*

- **While** and **case** statement can be combined to process arguments.

```
interactive=0
filename=~/sysinfo_page.html
while [ -n "$1" ]    # length > 0
do
   case $1 in
      -f | --file )           shift
                              filename=$1
                              ;;
      -i | --interactive )    interactive=1
                              ;;
      -h | --help )           printf "usage: %s [-f filename | -i | -h]\n" "$(basename $0)"
                              exit 0
                              ;;
      * )                     printf "usage: %s [-f filename | -i | -h]\n" "$(basename $0)"
                              exit 1
   esac
   shift
done
```

# Shell Scripting

### ***For*** **Loop**

- The for loop perform the given set of commands for each item in a list. It has the following syntax.

  **for** *var* in *<list>*

  **do**

      *<commands>*

  **done**

- Anything between **do** and **done** will be executed only once for each item in the list.
- Use **break** to break out of loop, **continue** for next iteration.

# Shell Scripting

***For*** **Loop**

```
names='Stan Kyle Cartman'
for name in $names
do
    echo $name
done
echo All done
```

- Outputs

```
Stan
Kyle
Cartman
All done
```

- Use break to break out of loop, continue for next iteration.

# Shell Scripting

## *For* Loop

- Can also use **for** loop to operate on list of files

  **for** name in *.txt
  **do**
      echo $name
  **done**
  echo All done

- Outputs

  file1.txt
  file2.txt
  file3.txt
  All done

- Use break to break out of loop, continue for next iteration.

# Shell Scripting

## *For* Loop

- Another variation similar to the C for loop has the following syntax.

```
for ((e1; e2; e3))
do
    <commands>
done
```

- Here is an example that echoes numbers between first and last. As with arithmetic expressions, variable references do not require '$':

```
first=5
last=10
for ((i=first; i <= last; i++))
do
    echo $i
done
```

- Use **break** to break out of loop, **continue** for next iteration.

# Shell Scripting

***Function***

- A function can have one of two equivalent forms

```
function_name() {
    <commands>
}
```

or

```
function function_name {
    <commands>
}
```

- Parameters are accessed by position: $1 for the first parameter, $2 for the second one, etc. $# is the number of parameters.
- The **shift** *n* operation shifts parameters left by *n*. The default value of *n* is 1.
- $0 is still the script name variable.

# Shell Scripting

***Function***

- The return value of a function is the same as the return/exit value of a command. 0 indicates success and other values indicate some other condition.
- Functions return 0 unless a **return** *n* statement is used, where *n* is the numeric code.  The return value of a function can be accessed through the special variable **$?** after calling the function.
- Functions can read from standard input and return results by writing them to standard output.  Functions can also participate in pipes with commands and other functions

# Shell Scripting

### *Function*

```
function  howdy {
    if [ $# -eq 0 ]
    then
        echo howdy stranger
    else
        echo howdy $1
    fi
}
howdy
howdy friend
echo "neighbor" | xargs howdy   # cannot call howdy from xargs
```

```
function  howdy {
    # if $1 not set, use "stranger"
    echo howdy ${1:-stranger}
}
```

- Output is

```
howdy stranger
howdy friend
xargs: howdy: no such file or directory
```

# Shell Scripting

***Function***

- Earlier we learned that a script and a compiled C program are treated equally when executed from the command line or within a script.
- Now we can see that within a script file, a function is also treated in exactly the same way as a script or a compiled C program, with only a few exceptions where a program or script name is required.
- Within a function the positional arguments access the arguments of the function. Outside of a function, the positional arguments refer to the arguments to the script.

# Shells and System Commands

***Function***

- Here is another example, word_counts that reads files and prints a frequency count of unique words in the specified files.

```
#!/bin/bash

# This script presents counts of unique words in input files in ascending order
# usage:  word_counts  [file] …

# words: breaks input into words and echoes them to output one per line
words() {
   while read -r line; do        # for each input line
      printf "%s\n" $line          # echo words one per line
    done
}

# pipes input files through words() for word list, then uniq and sort
cat $* | words | sort | uniq -c | sort -n
```

# Shell Scripting

**Variable Scopes**

- Bash supports scopes for variables. Unqualified variables are global to the script. It is also possible to declare variables local to a function by adding the qualifier **local**.

```
script_var=$1   # global variable has first script argument
function f {
      local local_var=$1  # local variable has first function argument
}
```

- Scripts can also export variables to make them visible globally in the environment that is running the shell script.

```
export global_var=$1   # exports the variable to the shell environment
```

- When the shell script that executes this export exits, global_var will be visible within that environment too, as well as to any other program or shell script run from that environment or this script.

# Shell Scripting

**Calling Functions in Other Scripts**

- Functions are local to the script where they are defined. A function in one script cannot be called directly from another script.
- However, there are two ways to accomplish the same thing:
  - Import script with functions into script that calls them
  - Build dispatch table in script with functions

# Shell Scripting

## Calling Functions in Other Scripts

- Here is how to import one script into another to call its functions.

  **second:**

  ```
  function pub_info {
      type=$1
      genre=$2
      printf "type=%s, genre=%s\n” "$type" "$genre"
  }
  ```

  **first:**

  ```
  source ./first
  pub_info novel mystery
  pub_info poetry romantic
  ```

- The second script in this example is located in the current directory.
- Be aware that the **source** command actually executes code outside of functions in the file specified by its argument.

# Shell Scripting

## Calling Functions in Other Scripts

- Another technique is to create a dispatch table in second.

**second-a:**

```
function pub_info {
    type=$1
    genre=$2
    printf "type=%s, genre=%s\n "$type" "$genre"
}
function second {
    case $1 in
    pub_info)     pub_info “$1” ”$2”
                  ;;
    *)            printf "Unknown selector: %s\n" "$1" 1>&2   # stderr
    esac
}
second $*
```

# Shell Scripting

**Calling Functions in Other Scripts**

- Now the functions in **second-a** can be called from **first-a** through the second() dispatch function in **second-a**.

  **first-a.sh:**
  ```
  second-a pub_info novel mystery
  second-a pub_info poetry romantic
  ```

- Note that this method is less efficient than including **second** in **first** because it requires starting a new process for each call to a function in **second-a**.