CSE 15L: Software Tools and Techniques Laboratory

Winter 2021 - http://ieng6.ucsd.edu/~cs15x

Instructors: Gary Gillespie Keith Muller

Class sessions will be recorded and made available to students asynchronously.

Schedule: Holiday on Monday!

Monday, February 15th is a Presidents Day

Schedule

Last Lecture

1. Shell Programming

Today

1. More Shell Programming!

Shell Logic Structures

- Four basic logic structures used in program development are:
  - *Sequential logic*

    To execute commands in the order in which they appear
  - *Decision Logic*

    To execute commands only if a certain condition is satisfied
  - *Looping Logic*

    To repeat a series of commands for given number of times
  - *Case Logic*

    To replace “if then/else if/else” statements when making numerous comparisons

Positional Parameters

When shell script or shell function is called with argument parameters, each is copied into special variables.

`$#` Number of input parameters

`$0` Name of the script

`$1`, **`$2`**, …, **`$9`** 1st, 2nd, and 9th argument parameter

`${1}`, **`${2}`**, …, **`${10}`** 1st, 2nd, and 10th argument parameter

`$@` List of input parameters

`$*` List of input parameters as space separated string

`shift` Shifts the positional parameters by one towards the beginning and drops **`$1`** from the list. After a **`shift`**, **`$2`** becomes **`$1`** and so on.

Note: If more than 9 arguments are used, they cannot be directly accessed by **`$1`** *to* **`$9`** *often the work around (or use ${} form) is to use the* **`shift`** *command!*

Positional Parameters and ${}

```
$ cat tricky.sh
echo $1 $10 ${10}
$ ./tricky.sh I II III IV V VI VII VIII IX
X XI
I I0 X
$
```

```
$ ls -a
.
..
out
test
$ vi makenewdir.sh
$ cat makenewdir.sh
#!/usr/bin/bash

if [ "$#" -ne 1 ]; then
    echo "Incorrect usage"
    exit 2
fi

dir_in="$1"
if [ -d "${dir_in}" ]; then
    echo "It exists. How nice."
    exit 1
else
    echo "Making ${dir_in}"
    mkdir "${dir_in}"
    exit 0
fi
```

For the script shown to the left, what is the expected output to the screen after running the following command?

```
$./makenewdir.sh
```

```
a. Incorrect usage

b. Incorrect usage
   2

c. It exists. How nice.

d. Making
   0
```

```
$ ls -a
.
..
out
test
$ vi makenewdir.sh
$ cat makenewdir.sh
#!/usr/bin/bash

if [ "$#" -ne 1 ]; then$
    echo "Incorrect usage"
    exit 2
fi

dir_in="$1"
if [ -d "${dir_in}" ]; then
    echo "It exists. How nice."
    exit 1
else
    echo "Making ${dir_in}"
    mkdir "${dir_in}"
    exit 0
fi
```

For the commands and outputs shown to the left, what is the expected output from the command `echo "$?"`?

```
$ ./makenewdir.sh src
$ echo "$?"
```

a. 0

b. 1

c. 2

d. None of the above

Looping Logic

- A loop is a block of code that is repeated several times. Either:
  - A pre-determined number of times determined by a list of items in the loop count (**`for`** loops), or
  - Until a particular condition is satisfied (**`while`** and **`until`** loops)

- To provide flexibility to the loop constructs there are also two statements for control:
  - **`continue`** : skips to the next item in a **`for`** loop
  - **`break`** : exits out of a loop

for each loops

A general form of a for loop:

```
for arg in list_args
do
     command1
     command2
     command3
done
```

where the value of variable **arg** is set to the values provided in **list_args** one at a time and the block of statements is executed. This is repeated until the list is exhausted.

Example 1: for loop

```
$ cat greettheoffice.sh
#!/usr/bin/bash
for person in Michael Jim Pam Dwight Creed
do
     echo Hello $person
done

$ ./greettheoffice.sh
Hello Michael
Hello Jim
Hello Pam
Hello Dwight
Hello Creed
```

Iterating through for loops

Common programming practice to iterate through index

```
for (int index = 0; index <= 10; index++)...
```

In bash script, we can use sequence expression `{istart..iend[..incr]}` where `[..incr]` is an optional step size

```
for index in {1..10}; do...
```

Example 2: **for** loop

```
$ cat oddoreven.sh
#!/usr/bin/bash

for index in {1..10}
do
     if [ $((index % 2)) = 1 ]
     then
          echo $index "is odd"
     else
          echo $index "is even"
     fi
done
```

Example 2: Syntax Error

```
$ cat oddoreven.sh
#!/usr/bin/bash

ind_max=10
for index in {1..${ind_max}}
do
      if [ $((index % 2)) = 1
            echo $index "is od
      else
            echo $index "is ev
      fi
done
```

!! Caution !!

Sequence expressions cannot use variables and will generate a syntax error

```
$ ./oddoreven.sh
./oddoreven.sh: line 7: {1..10}: syntax error:
operand expected (error token is "{1..10}")
```

Example 3: **for** loop

- Problem: take some set of actions for a given list of arguments.
  - You would like to be able to invoke your script like this:

    ./myscript *.txt
  - knowing that the shell will pattern match and build a list of filenames that match the *.txt pattern (any filename ending with *.txt*).

```
#!/usr/bin/bash
# change permissions on a bunch of files
for FN in $*
do
        echo changing $FN
        chmod 0750 $FN
done
```

Example 4: `for` loop

- Problem: dealing with embedded space in parameters

```
ls -ls "Oh the Waste"
0 -rw-r--r--  1 kmuller  staff  0 Feb 10 12:02 Oh the Waste
$ cat simpls.sh
# simple shell script
ls -l ${1}
$ ./simple.sh Oh the Waste
ls: Oh: No such file or directory
$ ./simpls.sh "Oh the Waste"
ls: Oh: No such file or directory
ls: the: No such file or directory
ls: Waste: No such file or directory
```

Fixed:

```
$ cat simpls.sh
# simple shell script
ls -l "${1} "
```

Example 3: **for** loop (revisited)

- Problem: take some set of actions for a given list of arguments.
  - You would like to be able to invoke your script like this:

    ./myscript *.txt

  - knowing that the shell will pattern match and build a list of filenames that match the *.txt pattern (any filename ending with *.txt*).

```
#!/usr/bin/bash
# change permissions on a bunch of files
for FN in "$@"
do
        echo changing "$FN"
        chmod 0750 "$FN"
done
```

Example 3: **for** loop (revisited)

- parameter $* expands to the list of arguments supplied to the shell script
- Consider a directory has MP3 files whose names are:

  vocals.mp3 cool music.mp3 tophit.mp3
- The second song title has a space in the filename between cool and music. When you invoke the script with:

  ```
  myscript *.mp3
  ```
- you'll get, in effect:

  ```
  myscript vocals.mp3 cool music.mp3 tophit.mp3
  ```
- If your script contains the line:

  ```
  for FN in $*
  ```
- it expands to

  ```
  for FN in vocals.mp3 cool music.mp3 tophit.mp3
  ```
- **$@** List of input parameters
- So, replacing that line in the script with:

  ```
  for FN in "$@"
  ```
- expands to

  ```
  for FN in "vocals.mp3" "cool music.mp3" "tophit.mp3"
  ```

Using Shift and loops

```
#!/usr/bin/bash
# use and consume an option
# parse the optional argument
VERBOSE=0
if [[ $1 = -v ]]
then
      VERBOSE=1
      shift
fi
# the real work is here
for FN in "$@"
do
      if (( VERBOSE == 1 ))
      then
             echo changing $FN
      fi
      chmod 0750 "$FN"
done
```

Iterating with variable conditions

Problem: How to iterate using variable conditions:

1. Use the seq command:

```
for ind in $(seq 1 $IND_MAX); do ...
```

seq [OPTION]... LAST

seq [OPTION]... FIRST LAST

seq [OPTION]... FIRST INCREMENT LAST

Print numbers from FIRST to LAST, in steps of INCREMENT.

2. Use ((...)) notation:

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

3. Use **`while`** loop

```
$ cat jaz.sh
#!/usr/bin/bash
counter=0
for ((num=0; num <= "$1"; num++))
do
  if (($num % 2 == 0))
  then
    echo -n "$num "
    counter=$((counter + 1))
  fi
done
echo "$counter"
```

For the script shown to the left, what is the output to the terminal after running:

```
$ ./jaz.sh 3
```

a. 2

b. 0 2

c. 0 2 2

d. 1 3

e. 1 3 2

More Complex Looping with a count

- You don’t need to use the $ construct (as in $i, except for arguments like $1) when referring to variables inside the double parentheses

```
for (( i=0, j=0 ; i+j < 10 ; i++, j++ ))
do
        echo $((i*j))
done
```

The **while** loop

A general form for a while loop:

```
while test-condition
do
    command1
    command2
done
```

- The **while** statement best illustrates how to set up a loop to test repeatedly for a matching condition like the if statement
- If the **test-condition** statement is true, the statements between **do** and **done** repeat

while loops

- Use the while looping construct for arithmetic conditions:

```
while (( COUNT < MAX ))
do
    some stuff
    let COUNT++
done
```

- for filesystem-related conditions

```
while [ -e "$LOCKFILE" ]
do
  some things
done
```

- For reading input (read returns 0 on success and 1 on end-of-file)

```
while read lineoftext
do
  process $lineoftext
done
```

Example 4: **while** loop

```
$ cat cumulativesum.sh
#!/usr/bin/bash

ind_max=$1
ind=1
sum=0

while [ "$ind" -le "$ind_max" ]
do
      echo Adding $ind into the sum.
      sum=`expr $sum + $ind`
      ind=`expr $ind + 1`
done
echo "The sum is $sum."
```

`until` loops

The syntax and usage is almost identical to `while` loops.

- Except that the block is executed until the test condition is satisfied (opposite of while loops!)

```
until test-condition
do
    command1
    command2
done
```

Note: You can think of until as equivalent to not_while

until example

- Script that attempts to copy a file to a directory and if it fails waits five seconds then tries again until it succeeds

```
until cp $1 $2
do
    echo 'Attempt to copy failed. waiting... '
    sleep 5
done
```

- written as a while

```
while ! cp $1 $2
do
    echo 'Attempt to copy failed. waiting... '
    sleep 5
done
```

Switch/Case Logic

- Switch logic structure simplifies the selection of a match when you have a list of choices
- It allows your program to perform one of many actions, depending upon the value of a variable without the use of extensive if/elif

case Statements

```
case argument in
   Pattern1)
         execute this if argument==Pattern1
         and this
         ;;
   Pattern2)
         execute this if argument==Pattern2
         ;;
esac
```

- Compares the string **argument** to listed patterns and executes code associated with the matching pattern.
- Matching starts with the first pattern and subsequent patterns are tested only if no earlier match is found.
- Default (catchall) is the `*)` case

Example 4: case statements

```
$ cat apple_vs_orange.sh
#!/usr/bin/bash
for index in {1..10}
do
     case $((index % 3)) in
           0)
                echo $index "apples"
                ;;
           1)
                echo $index "oranges"
                ;;
           2)
                echo $index "this code is silly"
                ;;
     esac
done
```

Example 5: **case** statements

```
case $FN in
         *.gif) gif2png $FN
               ;;
         *.png) pngOK $FN
                ;;
        *.jpg) jpg2gif $FN
                ;;
        *.tif | *.TIFF) tif2jpg $FN
               ;;
       *) echo "File not supported: " $FN
                ;;
esac
```

- The equivalent to this using if/then/else statements is:

```
if [[ $FN == *.gif ]]
then
               gif2png $FN
elif [[ $FN == *.png ]]
then
               pngOK $FN
elif [[ $FN == *.jpg ]]
then
               jpg2gif $FN
elif [[ $FN == *.tif || $FN == *.TIFF ]]
then
               tif2jpg $FN
else
               echo "File not supported:" $FN
fi
```

Revisiting shift: Accessing Input Arguments

Positional arguments only go from `$1` to `$9`. How do you access arguments past that?

METHOD 1

`for` loop with list of input arguments

```
for arg in "$@"; do
…
done
```

METHOD 2

loop through arguments by `shift`ing

```
while (("$#")); do
…
shift
…
done
```

Example 5: Processing Input Args

```
$ cat argloop.sh
#!/usr/bin/bash

for arg in "$@"
do
    echo "arg is now $arg"
done

$ ./argloop.sh This is a test
arg is now This
arg is now is
arg is now a
arg is now test
```

Example 6: Shift + Case + Input Processing

```
$ cat survey.sh
#!/usr/bin/bash
NAME=""
FAVLANG=""

while [ $# -ge 2 ]; do
    key="$1"
    val="$2"

    case "${key}" in
        "-n")
            NAME="$2" ;;
        "-f")
            FAVLANG="$2" ;;
        *)
            shift 1; continue ;;
    esac

    shift 2
done
echo "${NAME} likes to code in ${FAVLANG}"
```

By evaluating options/keys, programs can be run with arguments in any order!

```
./survey.sh -n Michael -f C
./survey.sh -f C -n Michael
```

both produce:

```
Michael likes to code in C
```

Functions

A general form for functions:

```
functionname()
{
        block of commands
}
```

- Functions group together commands so they can be executed via a single reference.
- Put any parameters for a *bash* function right after the function's name, separated by whitespace, just as if you were invoking any shell script or command.
  - Don't forget to quote them if necessary!
- To get values from a function you can assign values to variables inside the body of your function
  - Those variables will be global to the whole script
  - or use something like echo to send to standard output

Example 7: Functions

```
$ cat sumfunction.sh
#!/usr/bin/bash

sum() {
    xyz=`expr $1 + $2`
    echo $xyz
}

sum 5 3
echo "The sum of 4 and 7 is `sum 4 7`"
```

Next Lecture

1. XML and Ant