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.

Shell Programming

- *Shell variables*: Shell scripts support shell variables, which are symbolic names that can access values stored in memory
- *Operators*: Shell scripts support many operators, including those for performing mathematical operations
- *Logic structures*: Shell scripts support:
  - **Sequential Logic** for performing a series of commands
  - **Decision Logic** for branching from one point in script to another
  - **Looping Logic** for repeating a command several times
  - **Case Logic** for choosing an action from several possible options

The "Here" Document (heredoc)

- Sometimes you want to redirect standard input to come from a small section of text
- Rather than create a file with the text, use **<<** to create a temporary input source redirect text to come from the command line (or the rest of the shell script) up to the ending *marker (Heredoc)*
  - you can use any string as *marker* in place of HERE (e.g., EOF).

– Input stops when unique text (**HERE**) is read as a line by itself (Command line example)

```
$ cat << HERE
> This is text that will be redirected
> This text, too
> Redirection stops with HERE
> HERE
This is text that will be redirected
This text, too
Redirection stops with HERE
```

HEREDOC (shellscript)

```
$ cat ext.sh
#!/usr/bin/bash
grep $1 <<EOF
mike x.123
joe x.234
sue x.555
pete x.818
sara x.822
bill x.919
EOF

$ ./ext.sh mike
mike x.123
```

Use an ending marker with any escaping (' , “ or \) to turn off all scripting features in the HEREDOC

```
$ cat ext.sh
#!/usr/bin/bash
grep $1 <<EOF
# name amt
pete $100
joe $200
sam $ 25
bill $ 9
EOF

$ ./ext.sh mike
pete mike00
$ ./ext.sh bill
pete bill00
bill $ 9
$ ./ext.sh pete
pete pete00
```

```
$ cat ext.sh
#!/usr/bin/bash
grep $1 <<'EOF'
# name amt
pete $100
joe $200
sam $ 25
bill $ 9
EOF

$ ./ext.sh mike
$ ./ext.sh bill
bill $ 9
$ ./ext.sh pete
pete $100
```

Shell Operators

Shell Operators

- Bash shell operators are divided into three groups:
  - Defining and Evaluating operators
  - Redirecting and Piping operators
  - Arithmetic operators

Defining and Evaluating

- shell variable generalized form: **variable=value**

```
$ xyz=37; echo $xyz
37
$ unset xyz; echo $xyz

$
```

- You can set a pathname or a command to a variable or substitute to set the variable.

```
$ mydir=`pwd`; echo $mydir
$ mydir=$(pwd); echo $mydir
```

Review: Redirecting and Piping

- Redirecting via angle brackets **<, >, <<, >>**
  - Redirecting standard input and output follows a similar principle to that of piping except that redirects with files, not commands.
- Piping **|**
  - An important early development in Unix , a way to pass the output of one tool to the input of another.

EXAMPLE

```
$ touch file1                # ensure file1 is present
$ echo Hello > file1         # file1 got clobbered!

$ set –o noclobber           # prevent clobbering!
$ echo Hello > file1
-bash: file1: cannot overwrite existing file
$ echo Hello >> file1        # noclobber allows append!

$ set +o noclobber           # allow clobbering!
$ echo Hello > file1         # file1 clobbered again!
```

Arithmetic Operators

- If variables are strings, how do we do math?
- Several methods in bash!

  For the variable **`xyz=2020`**

  - **`expr`** command `$ xyz=`expr $xyz + 1``
  - **`let`** command `$ let "xyz=xyz + 1"`
  - **`(( … ))`** syntax `$ xyz=$((xyz + 1))`

- Note: All methods are for integer math only! If you need more precision, look up **`bc`**

Using **`expr`**

- **`expr`** command supports the following operators:
  - see man expr
  - arithmetic operators: +, -, *, /, %
  - comparison operators: <, <=, ==, !=, >=, >
  - Boolean/logical operators: &&, ||
  - parentheses: (…)
  - precedence is the same as C, Java

- **`expr`** takes values (numerical or evaluated variable **`$myvar`**) and operators as arguments

- **Caution!**
  - All arguments must be <u>separated by spaces</u>!!
  - To multiply, you must use * and not *

EXPR Example 1

```
num1=3
num2=5
op="+"
num3=`expr ${num1} ${op} ${num2}`
echo "Output is ${num3}"
Output is 8
(alternatively)
num3=`expr ${num1} + ${num2}`
echo "Output is ${num3}”
Output is 8
(but not)
num3=`expr ${num1}+${num2}`
echo "Output is ${num3}”
Output is 3+5
```

EXPR Example 2

```
num1=2
num2=8
num3=`expr num1+num2`
echo "The output is $num3"
```

```
The output is num1+num2
```

Using ((…)) and $((…))

- More operators than **expr** and they are more forgiving
- **((…))** executes math or logical operations then returns a status (via **$?**)
- **$((…))** executes ifand returns the value
- still no spaces around the = sign as with any bash variable assignment

- **Oddity:** $ normally put in front of a shell variable when we want its value is not needed inside the double parenthesis, the $ applies to the entire expression
- **Oddity Exception:** $ is needed for positional parameter (e.g., $2)

```
$ var=10
$ ((var++))
$ echo $?
0
$ echo $var
11
$ var2=$((var+10))          # no spaces needed around +
$ echo $var2
21
$ var2=((var+10))           # note $ is missing
-bash: syntax error near unexpected token `('
$ echo $?
1
```

Using Let

- let statement either requires quotes or that there be no spaces around not only the assignment operator (the equals sign), but any of the other operators as well; it must all be packed together into a single word

```
let i=2+2
let "i = 2 + 2"
```

```
COUNT=$((COUNT + 5 + MAX * 2))
let COUNT+='5+MAX*2'
```

If you use a positional parameter

```
COUNT=$((COUNT + $2 + OFFSET))
```

Operations with `let` and `((…))`

`var++ , var--`
`++var , --var`
	post/pre increment/decrement

`+ , -`	add, subtract
`* , / , %`	multiply/divide, remainder
`**`	power of
`= , += , -= , *= , /=`	assignment
`! , ~`	logical/bitwise negation
`& , | , ^`	bitwise AND, OR, XOR
`<< , >>`	left/right bitshift
`&& , ||`	logical AND, OR

Example: Arithmetic Operations

```
$ cat intcalc.sh
#!/usr/bin/bash
count=5
count=$((count++))
echo $count

$ chmod u+x intcalc.sh
$ ./intcalc.sh
6
```

Example: Arithmetic Operations

```
$ cat fpcalc.sh
#!/usr/bin/bash
a=5.48
b=10.32
c=`echo "$a + $b" | bc`
echo $c

$ chmod u+x fpcalc.sh
$ ./fpcalc.sh
15.80
```

Logic Structures

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

Comment: Command Return Values

- Properly written commands and built-ins return a value of 0 (zero) when they encounter no errors in their execution
- If commands and built-ins detect a problem (e.g., bad parameters, I/O errors, file not found), return some nonzero value (often a different value for each different kind of error they detect)

Decision Logic: Conditional

The simplest if construct
(the ; serves the same purpose as a newline, it ends a statement)

```
if test-command; then execute command; fi
```

The most general form of the **if** construct is:

```
if test-command
then
        execute command
elif test-command
then
        execute this
        and execute this command
else
        execute default command
fi
```

However, **elif** and/or **else** clause can be omitted.

What is a `test` command?

- `if` will examine exit status code `$?` of the `test-command`.
- With comparisons, `test` or `[ ]` command is commonly used
- [and test are both commands (try % which test or % which [)
  - bash now has built-ins for these (% type [or %type test)

- `test` is a command that returns exit status code of:

  ***true* (0)** or ***false* (1)** for a given set of arguments

- Argument structure is:

```
test item1 comparator item2
```

or

```
test option item1
```

- Instead of calling `test`, can use square brackets:

```
[ item1 comparator item2 ]
[ option item1 ]
```

NOTE: YOU MUST HAVE SPACES AROUND THE BRACKETS!!!

Using if and ((…))

```
$ var=10
$ if ((var == $var))
> then
> echo true
> fi
true
$
```

Comparators in `test`

`[ string1 = string2 ]`	True if strings are identical
`[ string1 == string2 ]`	…ditto….
`[ string1 != string2 ]`	True if strings are not identical
`[ string ]`	Return 0 exit status (=true) if string is not null
`[ -n string ]`	Return 0 exit status (=true) if string is not null
`[ -z string ]`	Return 0 exit status (=true) if string is null
`[ int1 –eq int2 ]`	Test identity
`[ int1 –ne int2 ]`	Test inequality
`[ int1 –lt int2 ]`	Less than
`[ int1 –gt int2 ]`	Greater than
`[ int1 –le int2 ]`	Less than or equal
`[ int1 –ge int2 ]`	Greater than or equal

Examples

```
# test a string to see if it has any length
if [ -n "$VAR" ]
then
    echo has text
else
   echo zero length
fi

if [ -z "$VAR" ]
then
    echo zero length
else
   echo has text
fi
```

- There are two types of variables that will have no length—those that have been set to an empty string and those that have not been set at all.
- This test does not distinguish between those two cases. All it asks is whether there are some characters in the variable.

if [] versus (( )) versus test

```
if [ $# -lt 3 ]
then
    echo "Error. Not enough arguments.\n"
    echo "usage: myscript file1 op file2\n"
    exit 1
fi
```

(alternatives)

```
if  ((  $#  <  3  ))
then
    echo "Error. Not enough arguments.\n"
    echo "usage: myscript file1 op file2\n"
    exit 1
fi

if test  $#  -lt 3
then
    echo "Error. Not enough arguments.\n"
    echo "usage: myscript file1 op file2\n"
    exit 1
fi
```

Examples: Decision Logic

SIMPLE EXAMPLE:

```
    if date | grep "Fri"
    then
        echo "It's Friday!"
    fi
```

FULL EXAMPLE:

```
    if [ "$1"  ==  "Monday" ]
    then
        echo "The typed argument is Monday."
    elif [ "$1" == "Tuesday" ]
    then
        echo "Typed argument is Tuesday"
    else
        echo "Typed argument is neither Monday nor Tuesday"
    fi
```

Note: `=` or `==` will both work in the test but `==` is better for readability.

test Options for File Inquiry

`[ -d filename ]`	Test if filename is a directory
`[ -f filename ]`	Test if filename is not a directory
`[ -s filename ]`	Test if filename has nonzero length
`[ -r filename ]`	Test if filename is readable
`[ -w filename ]`	Test if filename is writable
`[ -x filename ]`	Test if filename is executable
`[ -o filename ]`	Test if filename is owned by the user
`[ -e filename ]`	Test if filename exists
`[ -z filename ]`	Test if filename has zero length

All these conditions return true (0) if satisfied and false (1) otherwise.

All the file test conditions include an implicit test for existence

example: you do not need to test if a file exists and is readable, It won't be readable if it doesn't exist.

Common Errors with **`test`**

Less than/Greater than:

- When evaluating, bash will interpret <, <=, >, >= as standard IO redirection!

```
$ if [ ind < 10 ]; then
> echo $ind
> ((ind++))
> fi
bash: 10: No such file or directory
```

```
$ if [ ind <= 10 ]; then
> echo $ind
> ((ind++))
> fi
bash: =: No such file or directory
```

- To fix, use **`-ge`**, **`-gt`**, **`-lt`**, **`-le`** or **`\<  \>`**

Common Errors with **test**

Comparing space separated strings:

- When evaluating strings, remember that each space-separated word is a new argument

```
$ input="My file.txt"
$ if [ $input == "my_file.txt" ]; then
> data=`cat "$input"`
> fi
bash: [: too many arguments
$ echo [ $input == "my_file.txt" ]
[ My file.txt == my_file.txt ]
```

- To fix, use quotes around arguments!

Common Errors with `test`

Comparing variables with no set value:

- Instead of a comparison between two values, expression will revert to comparisons of strings/files with single variable

```
$ unset var
$ echo $var
                (space)
$
$ if test $var != 10; then  # note no quotes around $var
    > echo var is not 10
    > fi
-bash: [: ==: unary operator expected

$ echo test $var != 10
test != 10
$ echo [ $var != 10 ]
[ != 10 ]
```

- Stay safe with comparisons! Put variables in quotes or use `[[ … ]]` for bash-specific interpretation

What is [[ ]] ?

- [[ ]] has some special behavior that [doesn't have
  - You no longer must quote variables
    because [[handles empty strings and strings with whitespace more intuitively.
- For example, with [you must write

```
if [ -f "$file" ]
```

to correctly handle empty strings or file names with spaces in them.

With [[the quotes are unnecessary:

```
if [[ -f $file ]]
```

Common Errors with **test**

```
$ unset var
$ if test $var != 10; then       # note no quotes around $var
    > echo var is not 10
    > fi
-bash: [: ==: unary operator expected
    $ if test "$var" != 10; then # note quotes around $var
    > echo var is not 10
    > fi
var is not 10
    $ if [ $var != 10 ]; then     # note no quotes around $var
    > echo var is not 10
    > fi
-bash: [: ==: unary operator expected
    > if [ "$var" != 10 ]; then   # note quotes around $var
    > echo var is not 10
    > fi
    var is not 10
    $ if [[ $var != 10 ]]; then   # note no quotes around $var
    > echo var is 10
    > fi
    var is not 10
```

test and []: YOU NEED TO BE CAREFUL OF NULL VARIABLES!!!

Example: Decision Logic

A simple example

```
#!/usr/bin/bash
if [ "$#" -ne 2 ]
then
   echo $0 needs two parameters!
   echo You are inputting $# parameters.
         exit 2
else
   par1=$1
   par2=$2
fi
echo $par1
echo $par2
```

Example: Decision Logic

Another example:

```
#!/usr/bin/bash
#  number is positive, zero or negative
echo –n "enter a number:"
read number
if [ "$number" -lt 0 ]
then
   echo "negative"
elif [ "$number" -eq 0 ]
then
   echo zero
else
   echo positive
fi
```

Use the -eq operator for numeric comparisons and the equality primary = (or ==) for string comparisons

```
 #!/usr/bin/bash
VAR1=" 05 ”
VAR2="5"
echo –n "do they -eq as equal? "
if [ "$VAR1" -eq "$VAR2" ]
then
        echo YES
else
        echo NO
fi
echo –n "do they = as equal? ”
if [ "$VAR1" = "$VAR2" ]
then
        echo YES
else
        echo NO
fi
```

When we run the script, here is what we get:

```
do they -eq as equal? YES
do they = as equal? NO
$
```

Combining Tests

`&&` represents AND

`||` represents OR

Syntax:

```
if cond1 && cond2 || cond3 …
```

An alternative form is to use a compound statement in **`test`** using **`—a`** and **`—o`** keywords. For example:

```
if [ cond1 —a cond2 —o cond3 … ]
```

where **`cond1`**, **`cond2`**, **`cond3`** are either commands returning a value or test conditions of the form:

`test arg1 op arg2` or **`[ arg1 op arg2 ]`**

Example: Combining Tests

```
if  `date | grep "Fri"` && `date +'%H'` -gt 17
then
      echo "It’s Friday, it’s home time!!!"
else
      echo not Friday
fi


# note the spaces around ] and [
if [ "$a" –lt 0 –o "$a" –gt 100 ]; then
      echo "limits exceeded"
fi
```

Combining Tests

- You might need to use parentheses to get the proper precedence, as in a and (b or c),
- When you use parentheses, be sure to escape their special meaning from the shell by putting a backslash before each or by quoting each parenthesis.

```
if [ -r "$FN" -a \( -f "$FN" -o -p "$FN" \) ]
```

- In C and Java, if the first part of the AND expression is false (or the first part true in an OR expression), the second part of the expression won't be evaluated (this is called an expression *short-circuit*).
- However, because the shell makes multiple passes over the statement while preparing it for evaluation (e.g., doing parameter substitution, etc.), both parts of the joined condition may have been partially evaluated
- **Summary:** do not count on short circuits!