PURDUE
UNIVERSITY®

# CS 25200: Systems Programming

# Lecture 3: More Bash Scripting

Prof. Turkstra

# Lecture 03

- More Bash
  - Reading and writing
  - Control loops
  - Decision making
  - Arrays
- I/O redirection
- Quotes
- grep
- Basic regular expressions

# `printf` command - formatted output

- Almost like in C:
  `printf "format" list of variables`
  - Format string is like that found in C
  - The list of variables should be separated by spaces.
- Example:

```
A=3.45
S="Big Deal"
printf "A: %5.2f, S: %s\n" $A "$S"
```

# More Bash variables

- Untyped variables can be used to hold strings, integers, and floating point values

```
#! /bin/bash
A="Big Deal"
echo $A
A=3
echo $A
A=2.3
echo $A
exit 0
```

Output:

```
Big Deal
3
2.3
```

# Typed variables

- Integers
  `typeset -i variable_name`
- Cannot be used to store strings

# Example

```
#! /bin/bash
typeset -i A
A=3
echo "A: $A"
A="Oh Well"
echo "A: $A"
A=2.4
echo "A: $A"
exit 0
```

Results in the following output:

```
A: 3
X1: line 5: Oh Well: syntax error in expression (error token is
   “Well”)
A: 3
X1: line 7: 2.4: syntax error: invalid arithmetic operator (error
   token is “.4”)
A: 3
```

# Floating point variables

- Bash does not support floating point values or math. :-(

# Constants

- We can create read only variables too...

```
typeset -r Name="Ffej"
typeset -r K=8
```

- Or how about a read only integer...

```
typeset -ri K=8
```

# Example

```
#! /bin/bash
typeset -r Name="Ffej Artskrut"
echo $Name
Name="Big Deal"
echo $Name
exit 0
```

Results in the following:

```
$ C
Ffej Artskrut
./C: line 4: Name: readonly variable
Ffej Artskrut
```

# `for` loops

- Two different syntaxes in Bash

```
for variable in list
do
  commands
done
```

where `list` is a set of strings separated by whitespace

```
for (( initial_expression; loop_condition;
       loop_expression ))
do
  commands
done
```

- `do` must be on the 2nd line in both cases!

# Examples

```
for I in 1 2 3 4 5
do
  echo -n ${I}
done

for (( I = 1; I < 6; I++ ))
do
  echo -n ${I}
done
```

Both result in the same output:

```
12345
```

# Examples cont...

```
#! /bin/bash
ls *.c > temp:$$
touch temp:$$
for File in $(cat temp:$$)
do
  lp -dsomeprinter ${File}
done
rm -f temp:$$
exit 0
```

# `while` loop

- Note that a command is considered "true" as long as its exit status is zero. This is at times backwards from what you might be used to.

```
while <command exit status is true (0)>
do
  <whatever commands you need to do the job>
done
```

# Example

```
#! /bin/bash
typeset -i I=0
while (( I < 10 ))
do
  echo -n " ${I}"
  (( I = I + 1 ))
done
echo
exit 0
```

Outputs:

```
0 1 2 3 4 5 6 7 8 9
```

# `read` command

- read can be used to obtain user input. Eg,

```
echo -n "Feed me data: "
read DataVar
echo "You fed me ${DataVar}!"
```

- It may also be used to read from other streams such as a *file*.
  - Everyone always forgets that the name of the file goes at the *end* of the while loop

# reading from a file

```
#! /bin/bash
cat InFile
echo
while read A B C
do
  echo "A: |${A}|" \
       "B: |${B}|" \
       "C: |${C}|"
done < InFile
exit 0
```

# Output

- This is the output generated from the `cat` command, which simply displays a file's contents to stdout:

```
This is Neat!
CS 252 is Great!
Ffej is the best.
1 2 3 4 5 6
Big Deal
1234
```

- This is the output generated by the `echo` statement inside of the `while` loop:

```
A: |This| B: |is| C: |Neat!|
A: |CS| B: |252| C: |is Great!|
A: |Ffej| B: |is| C: |the best.|
A: |1| B: |2| C: |3 4 5 6|
A: |Big| B: |Deal| C: ||
A: |1234| B: || C: ||
```

# Bash math

- Bash only has integer math.
- Use `let` or `(( … ))` to isolate mathematical statements
- Basic math operators include: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulus (`%`)

# Integer math example

```
#! /bin/bash
typeset -i a=11
typeset -i b=3
typeset -i x
(( x = a + b ))
echo "(( x = $a + $b )) x = $x"
(( x = a - b ))
echo "(( x = $a - $b )) x = $x"
(( x = a * b ))
echo "(( x = $a * $b )) x = $x"
(( x = a / b ))
echo "(( x = $a / $b )) x = $x"
(( x = a % b ))
echo "(( x = $a % $b )) x = $x"
exit 0
```

# Output

```
(( x = 11 + 3 )) x = 14
(( x = 11 - 3 )) x = 8
(( x = 11 * 3 )) x = 33
(( x = 11 / 3 )) x = 3
(( x = 11 % 3 )) x = 2
```

# Branching

- ```
  if command_true
  then
    commands
  else
    commands
  fi
  ```
- Note: the command is "true" if it returns 0 - neat!


  ```

# Nested `if` statements

- As one might expect, it is possible to nest branches in Bash...

```
if (( A < B ))
then
  echo "A < B"
else
  if (( A == B ))
  then
    echo "A = B"
  else
    echo "A > B"
  fi
fi
```

# `elif` statement

- Equivalent to "else if" in C

```
if (( A < B ))
then
  echo "A < B"
elif (( A == B ))
then
  echo "A = B"
else
  echo "A > B"
fi
```

# Testing

- Tests of any sort should be surrounded by double brackets with spaces:
  `[[ space whatever space ]]`
- Example

```
if [[ -r MyFile ]]
then
  echo MyFile is readable!
fi
```

- Tests include logical comparisons, string comparisons, and file permission tests

# File testing

| | |
|---|---|
| `-a` | file exists |
| `-d` | is a directory |
| `-f` | is an ordinary file |
| `-r` | is readable |
| `-s` | has non-zero length |
| `-w` | is writable |
| `-x` | is executable |
| | …and lots more |
| `!` | reverse the test |

# Arithmetic comparison

- Comparisons between numbers should always be surrounded by double parentheses:

```
(( whatever ))
```

- For example:

```
if (( 7 <= 5 ))
then
  echo "The world is at an end!"
fi
```

- Arithmetic comparisons include:

```
==  equal >=    greater than or equal
 >  greater than      <=    less than or equal
 <  less than   !=    not equal
```

# String tests

- Equality,

```
if [[ string1 = string2orpattern ]]
if [[ string1 == string2orpattern ]]
if [[ string1 != string2orpattern ]]
```

- Lexicographical ordering,

```
if [[ string1 < string2 ]]
if [[ string1 > string2 ]]
```

- Emptiness,

```
if [[ -n string1 ]] # string is not NULL
if [[ -z string1 ]] # string is NULL
```

# Example

```
#! /bin/bash
#

if (( $# != 1 )); then
  echo "Usage: $0 <filename>"
  exit 1
fi
File="$1"
if [[ ! -f "${File}" ]]; then
  echo "File: ${File} is not an ordinary file"
else
  echo "File: ${File} is an ordinary file"
fi
exit 0
```

# Output

```
$ File_Check
Usage: File_Check <filename>

$ File_Check x
File x is not an ordinary file

$ File_Check File_Check
File File_Check is an ordinary file
```

# Purdue trivia

- "While students, future author George Ade and cartoonist John McCutcheon faced the wrath of President James H. Smart for attending a ladies' literary society meeting without faculty permission, and McCutcheon got fired as editor of the student newspaper."

  - A Century and Beyond, by Robert W. Topping

# Arrays in Bash

- Unlike C, bash supports sparse arrays

```
ArrVar[5]=8
ArrVar[15]=12
ArrVar[19]=7
```

- If the indices aren't consecutive, how do we know the array's size? How do we know the indices for all values?

# Special operators # and !

- You can obtain a list (a string of whitespace-separated values) of every element in an array:

```
echo ${ArrVar[*]}
echo ${ArrVar[@]}
```

- The size of an array can be found by using the # operator:

  `${#ArrVar[*]}` or `${#ArrVar[@]}`

- The array subscripts can be found by using the ! operator:

  `${!ArrVar[*]}` or `${!ArrVar[@]}`

# Indexed array example

```
#! /bin/bash
A[5]=34
A[1]=3
A[2]=56
A[100]=89
echo "Size of array: ${#A[*]}"
echo "Array indices: ${!A[*]}"
for I in ${!A[*]}
do
  echo "A[${I}]=${A[I]}"
done
exit 0
```

# Indexed array output

```
Size of array: 4
Array indices: 1 2 5 100
A[1]=3
A[2]=56
A[5]=34
A[100]=89
```

# Read array example

```
#! /bin/bash
echo "Data_File:"
cat Data_File       # Remember, cat just dumps the file's
echo                # contents to stdout
echo "Formatted output:"
while read -a Data  # Split on whitespace
do                  # (spaces and tabs)
  for (( I = 0; I < ${#Data[*]}; I++ ))
  do
    printf "%6.2f" ${Data[I]}
  done
  echo
done < Data_File
exit 0
```

# Output

```
Data_File:
1 2 3                  77
12 12.6 6.8          7
2 1.0             -3   -5.5
```

Formatted output

```
 1.00  2.00  3.00 77.00
12.00 12.60  6.80  7.00
 2.00  1.00 -3.00 -5.50
```

# Binary operators

- You remember our friends from CS 240, right?

| | |
|---|---|
| `<< n` | Shift left `n` bits |
| `>> n` | Shift right `n` bits |
| `&` | Bitwise AND |
| `^` | Bitwise EXCLUSIVE OR |
| `\|` | Bitwise OR |
| `~` | Bitwise negation |

# Binary operations

```
#! /bin/bash
typeset -i A=2#1101
typeset -i B=2#0110
typeset -i C
(( C = A & B ))
echo "(( C = $A & $B )) C = $C"
(( C = A | B ))
echo "(( C = $A | $B )) C = $C"
(( C = A ^ B ))
echo "(( C = $A ^ $B )) C = $C"
exit 0
```

Results (the values actually displayed are in base 10):

```
(( C = 2#1101 & 2#110 )) C = 2#100
(( C = 2#1101 | 2#110 )) C = 2#1111
(( C = 2#1101 ^ 2#110 )) C = 2#1011
```

# More binary operations

```
#! /bin/bash
typeset -i A=2#1101
typeset -i B=2#0110
typeset -i C
(( C = A << 2 ))
echo "(( C = $A << 2 )) C = $C"
(( C = B >> 1 ))
echo "(( C = $B >> 2 )) C = $C"
(( C = ~B ))
echo "(( C = ~$B )) C = $C"
exit 0
```

Results in:

```
(( C = 2#1101 << 2 )) C = 2#110100
(( C = 2#110 >> 1 )) C = 2#11
(( C = ~2#110 )) C = 2#11111111111111111111111111111001
```

# More on loops

- Similar to C, bash has...

  `continue n`

  - Used to stop the execution of the innermost `n` loops and then continue with the next loop. The default is `n = 1`.

  `break n`

  - Used to end the execution of the innermost `n` loops. Default is `n = 1`.

# Examples

```
#! /bin/bash
for (( I = 0; I <= 4; I++ )); do
  if (( I == 1 )); then
    continue
  fi
  echo -n " ${I}"
done
echo
exit 0
```

Results in the following output:

```
0 2 3 4
```

# Examples cont...

```
#! /bin/bash
I=0
while (( I <= 4 )); do
  if (( I == 1 )); then
    break
  fi
  echo -n " ${I}"
  (( I++ ))
done
echo
exit 0
```

Results in the following:

```
0
```

# What about arguments?

- What if we want to loop through the command line arguments?
- Even if we know how many there are, we still can't use a loop construct...

```
#! /bin/bash
for (( I = 0; I < $#; I++ )); do
  echo $what???
done
```

# shift n

- Used to left shift the parameters on the command line `n` places
  - Default is `n = 1`
- `$0` is never changed
- Often used when an unknown number of parameters are passed, or for looping through a large number of parameters.

# Example

```
#! /bin/bash
echo '$0 -- ' $0
echo '$# -- ' $#
X=0
while (( $# != 0 )); do
  (( X = X + 1 ))
  echo "\"\${X}\" was $1"
  shift
done
exit 0
```

Sample run...

```
$ parameters q "1 2 3" xyz
$0 -- parameters
$# -- 3
"$1" was q
"$2" was 1 2 3
"$3" was xyz
```

# I/O redirection - reading

- To redirect input for a program or command,

  `< file` `n < file` n is the file descriptor

  `<< file` `n << file` n is the file descriptor
- Example:

```
mail jeff@purdue.edu < my_document
```

# I/O redirection - writing

- We can redirect the output from a program or command too!
  - `> file` and `n > file` Redirect output to `file`
  - `>> file` and `n >> file` Appends output to `file`
  - `>| file` and `n >| file` Overrides the noclobber option, if set
  - `>& number` Redirects the output to file descriptor `number`

# Examples

```
#! /bin/bash

exec 3< $1
exec 4> $2

I=0

while read <&3 Line; do
  echo "Line ${I}: ${Line}" >&4
done

exec 3>&-
exec 4>&-

exit 0
```

# Questions?