INTRODUCTION TO SHELL SCRIPTING

PART 2

Dr. Jeffrey Frey
University of Delaware, IT

Dare to be first.

UNIVERSITY OF DELAWARE

version 2

GOALS — PART 2

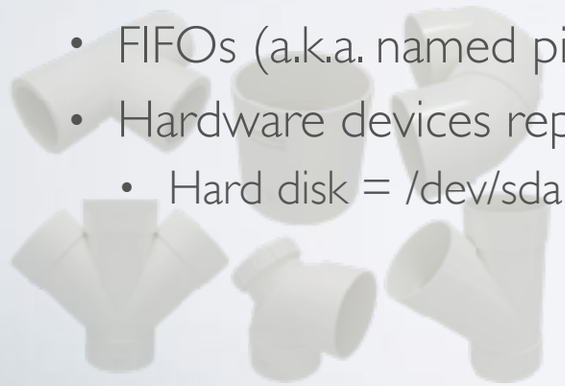
- Shell plumbing review
  - Standard files
  - Redirection
  - Pipes

GOALS — PART 2

- Command substitution
  - backticks vs. dollar–parenthesis
- Advanced variables
  - arrays
- Advanced language constructs
  - subroutines
  - case statements (branch tables)
  - and/or lists

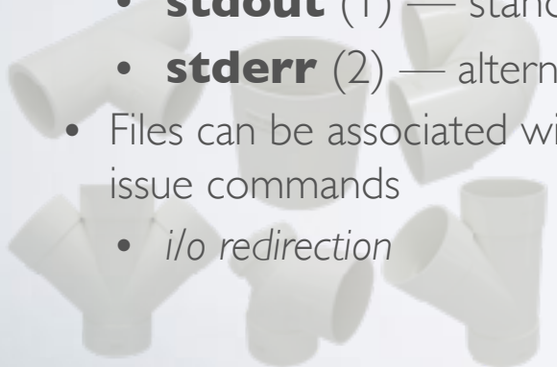
PLUMBING REVIEW

- In Unix/Linux, "*file*s" play a central role
  - Regular files (sequence of N bytes)
  - Directories (collection of file entities)
  - Sockets (e.g. network programming)
  - FIFOs (a.k.a. named pipes)
  - Hardware devices represented as files
    - Hard disk = /dev/sda



PLUMBING REVIEW

- Programs perform i/o by opening, reading, writing files
  - Three standard files open for every program:
    - **stdin** (0) — standard input (e.g. from keyboard)
    - **stdout** (1) — standard output (e.g. to terminal)
    - **stderr** (2) — alternate output (debugging, errors)
  - Files can be associated with the standard channels when you issue commands
    - *i/o redirection*



PLUMBING REVIEW

- Take program input **stdin** from a file

```
[frey@mills ~]$ ./my_program < input.txt
```

- Send output to **stdout** to a file
  - Truncate file

```
[frey@mills ~]$ ./my program > output.txt
```

  - Append to existing file (create if it doesn't)

```
[frey@mills ~]$ ./my program >> output.txt
```

PLUMBING REVIEW

- Send alternate output (**stderr**) to a file
  - Truncate file

```
[frey@mills ~]$ ./my_program 2> output.txt
```

  - Append to existing file (create if it doesn't)

```
[frey@mills ~]$ ./my program 2>> output.txt
```

  - Send to same file as **stdout**

```
[frey@mills ~]$ ./my_program 2>&1
```

PLUMBING REVIEW

- Send alternate output (**stderr**) to a file
  - Order is important

```
[frey@mills ~]$ ./my_program > output.text 2>&1
```

    - Close file #1, open output.text as file #1
    - Also open file #1 as file #2

```
[frey@mills ~]$ ./my program 2>&1 > output.text
```

    - Also open file #1 as file #2
    - Close file #1, open output.text as file #1

In first case, **stderr** and **stdout** both get written to "output.text." In the second case, **stderr** continues being written to e.g. terminal while **stdout** is written to "output.text."

- Ser
  - Order is important

```
[frey@mills ~]$ ./my_program > output.text 2>&1
```

    - Close file #1, open output.text as file #1
    - Also open file #1 as file #2

```
[frey@mills ~]$ ./my program 2>&1 > output.text
```

    - Also open file #1 as file #2
    - Close file #1, open output.text as file #1

PLUMBING REVIEW

- Reuse output from one program as input for another
  - Write to a temporary file

```
[frey@mills ~]$ ./my program > /tmp/output.text
```

  - Read from the temporary file

```
[frey@mills ~]$ ./my_other_program < /tmp/output.text > other_output.text
```

  - Remove temporary file

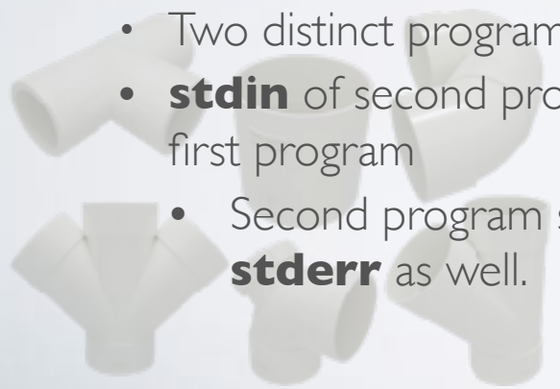
```
[frey@mills ~]$ rm /tmp/output.text
```

PLUMBING REVIEW

- Reuse output from one program as input for another
  - Pipes simplify this action

```
[frey@mills ~]$ ./my program | ./my other program > other output.text
```

    - Two distinct programs are executed
    - **stdin** of second program is "connected" to **stdout** of first program
      - Second program sees just **stdout** from first, not **stderr** as well.



PLUMBING REVIEW

- Reuse output from one program as input for another
  - Pipes simplify this action

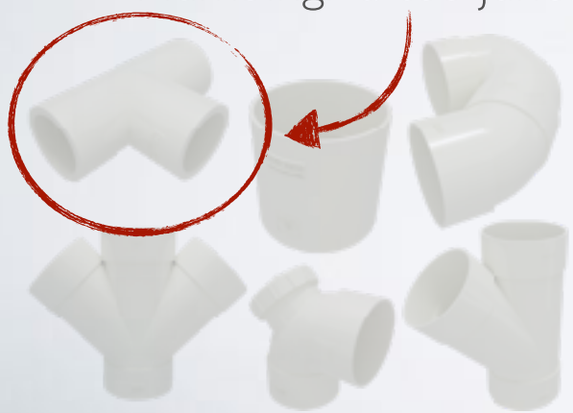
```
[frey@mills ~]$ ./m
```

Unless I do what to the first program's stderr?

  - Two dis
  - **stdin** of first pro
    - Second program sees just **stdout** from first, not **stderr** as well.

PLUMBING REVIEW

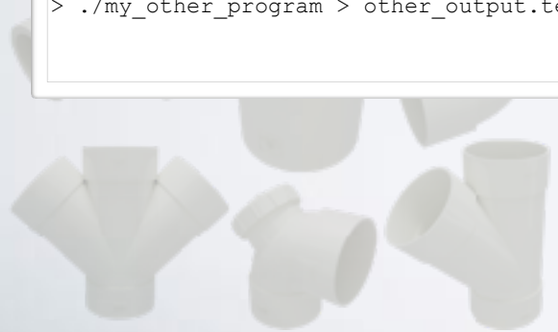
- Duplicate output across two files
  - Pipe output to another program but also write it to a file
  - Plumbing: a tee-junction



PLUMBING REVIEW

- Duplicate output across two files
  - Pipe output to another program but also write it to a file
  - Plumbing: a tee-junction

```
[frey@mills ~]$ ./my_program | tee output.text | \
> ./my_other_program > other_output.text
```



PLUMBING REVIEW

- Special files to use in redirection

	…as stdin	…as stdout
/dev/null	empty input	discard all output
/dev/zero	infinite sequence of zero-valued bytes	—
/dev/random	sequence of pseudo-random bytes	—
/dev/urandom	randomer pseudo-random bytes	—

COMMAND SUBSTITUTION

- Execute command:

```
[frey@mills ~]$ ls -1 /dev/disk/by-id | grep ^scsi | sed 's/ ->.*$//'
scsi-36c81f660f995ef001b57cce024739130
scsi-36c81f660f995ef001b57cce024739130-part1
scsi-36c81f660f995ef001b57d3cc0e25b769
scsi-36c81f660f995ef001b57d3cc0e25b769-part1
scsi-36c81f660f995ef001b57d3cc0e25b769-part2
scsi-36c81f660f995ef001b57d3cc0e25b769-part3
```

COMMAND SUBSTITUTION

- Assign **stdout** of command to variable:

```
[frey@mills ~]$ DISKS=`ls -1 /dev/disk/by-id | grep ^scsi | sed 's/ ->.*$//'`

[frey@mills ~]$ echo $DISKS
scsi-36c81f660f995ef001b57cce024739130 scsi-36c81f660f995ef001b57cce024739130-part1
scsi-36c81f660f995ef001b57d3cc0e25b769 scsi-36c81f660f995ef001b57d3cc0e25b769-part1
scsi-36c81f660f995ef001b57d3cc0e25b769-part2
scsi-36c81f660f995ef001b57d3cc0e25b769-part3
```

OR

```
[frey@mills ~]$ DISKS=$(ls -1 /dev/disk/by-id | grep ^scsi | sed 's/ ->.*$//')

[frey@mills ~]$ echo $DISKS
scsi-36c81f660f995ef001b57cce024739130 scsi-36c81f660f995ef001b57cce024739130-part1
scsi-36c81f660f995ef001b57d3cc0e25b769 scsi-36c81f660f995ef001b57d3cc0e25b769-part1
scsi-36c81f660f995ef001b57d3cc0e25b769-part2
scsi-36c81f660f995ef001b57d3cc0e25b769-part3
```

COMMAND SUBSTITUTION

- Backticks
  - Must be careful to escape some characters
    - \$ \` \\
  - Hard to nest substitutions
- Dollar-parentheses
  - Text inside parentheses used verbatim
  - Easy to nest substitutions

```
[frey@mills ~]$ DISKS=$(ls -1 $(echo /dev/disk))
```

COMMAND SUBSTITUTION

- Dollar-parentheses

```
[frey@mills ~]$ DISKS=$(ls -1 $(echo /dev/disk/by-id) | grep ^scsi | sed 's/ -
>.*$//')

[frey@mills ~]$ echo $DISKS
scsi-36c81f660f995ef001b57cce024739130 scsi-36c81f660f995ef001b57cce024739130-part1
scsi-36c81f660f995ef001b57d3cc0e25b769 scsi-36c81f660f995ef001b57d3cc0e25b769-part1
scsi-36c81f660f995ef001b57d3cc0e25b769-part2
scsi-36c81f660f995ef001b57d3cc0e25b769-part3
```

COMMAND SUBSTITUTION

- Newlines will be replaced by a simple space…
  - …unless the substitution is inside double quotes:

```
[frey@mills ~]$ DISKS="$(ls -1 /dev/disk/by-id | grep ^scsi | sed 's/ ->.*$//')"

[frey@mills ~]$ printf "%s\n" "$DISKS"
scsi-36c81f660f995ef001b57cce024739130
scsi-36c81f660f995ef001b57cce024739130-part1
scsi-36c81f660f995ef001b57d3cc0e25b769
scsi-36c81f660f995ef001b57d3cc0e25b769-part1
scsi-36c81f660f995ef001b57d3cc0e25b769-part2
scsi-36c81f660f995ef001b57d3cc0e25b769-part3
```

LOOP OVER WORDS

- Use the "for" loop over a set of words
  - Words come from a command substitutions

```
[frey@mills ~]$ \
> for disk in $(ls -1 /dev/disk/by-id | grep ^scsi | sed 's/ ->.*$//'); do
>   if [[ $disk =~ ^scsi-([0-9a-f]+)$ ]]; then
>     echo "SCSI disk found, id = ${BASH_REMATCH[1]}"
>   fi
> done

SCSI disk found, id = 36c81f660f995ef001b57cce024739130
SCSI disk found, id = 36c81f660f995ef001b57d3cc0e25b769
```

LOOP OVER WORDS

- Use the "fo
  - Words co

Any guess what this variable syntax signifies?

```
[frey@mills ~]$ \
> for disk in $(ls
>   if [[ $disk =~ ^scsi-([0-9a-f]+)$ ]]; then
>     echo "SCSI disk found, id = ${BASH_REMATCH[1]}"
>   fi
> done

SCSI disk found, id = 36c81f660f995ef001b57cce024739130
SCSI disk found, id = 36c81f660f995ef001b57d3cc0e25b769
```

ADVANCED EXPANSION

- Variable names can also be enclosed within curly braces: `${PREFIX}`
- Curly braces allow for additional logic and transformation w.r.t. the variable's value

ADVANCED EXPANSION

${!VAR}

```
> echo $PATH
/usr/bin:/bin:/opt/bin:…

> echo $VAR
PATH

> echo ${!VAR}
/usr/bin:/bin:/opt/bin:…
```

- Indirect expansion
- The value of $VAR is itself the name of a variable
- Substitute the value of *that* variable

ADVANCED EXPANSION

`${!VAR[*@]}`

```
> VERBOSE=1
> VARIABLE=1
> echo ${!V*}
VAR VARIABLE VERBOSE
> echo ${!VAR*}
VAR VARIABLE
```

- Names of variables whose name start with "VAR"
- * versus @ has usual significance w.r.t. parsing

ADVANCED EXPANSION

`${VAR:[-=?+]word}`

```
> unset VAR
> echo ${VAR:?not set}
-bash: VAR: not set
> echo ${VAR:+xyz}

> echo ${VAR:-xyz}
xyz
> echo $VAR

> echo ${VAR:=xyz}
xyz
> echo ${VAR:+is set}
is set
> echo $VAR
xyz
```

- Provide actions if VAR is empty:

-	use *word*
=	assign *word* to $VAR and use *word*
?	display an error and *word*

- …or if VAR is not empty:

+	use *word*

- "Empty" means not set or null-valued

ADVANCED EXPANSION

`${VAR[-=?+]word}`

```
> unset VAR
> echo ${VAR:-xyz}
xyz
> echo ${VAR-xyz}
xyz

> VAR=
> echo ${VAR:-xyz}
xyz
> echo ${VAR-xyz}

>
```

- Provide actions if VAR is not set:

-	use *word*
=	assign *word* to $VAR and use *word*
?	display an error and *word*

- …or if VAR is set:

+	use *word*

- Leaving the colon out, a null-valued variable behaves as though it has a value

ADVANCED EXPANSION

${#VAR}

```
> echo ${#VAR}
3
> demo() { echo ${#@}; }
> demo a b c
3
> demo "a b" c
2
```

- Number of characters in the value of $VAR
- If VAR is "*" or "@", the number of arguments to the script/function

ADVANCED EXPANSION

`${VAR:offset[:length]}`

```
> echo ${VAR:0}
xyz
> echo ${VAR:1}
yz
> echo ${VAR:1:1}
y
> echo ${VAR:0:${#VAR}-1}
xy
> echo ${VAR: -2}
yz
```

- Use a substring of the value of $VAR
- Without *length*, all characters from *offset* to the end
- First character is *offset* zero
- Negative *offset* is relative to end of string

- Space is necessary because ":-" is used to indicate default value

ADVANCED EXPANSION

`${VAR:offset[:length]}`

```
> echo ${VAR:0}
xyz
> echo ${VAR:1}
yz
> echo ${VAR:1:1}
y
> echo ${VAR:0:${#VAR}-1}
xy
> echo ${VAR: -2}
yz
```

Why is there a space here?

- Use a substring of the value of $VAR
- Without *length*, all characters from *offset* to the end
- First character is *offset* zero
- relative to

• Space is necessary because ":-" is used to indicate default value

ADVANCED EXPANSION

${VAR#{#}*word*}

```
> echo ${VAR#x}
yz
> VAR=xxxyyzzz
> echo ${VAR#x}
xxyyzzz
> echo ${VAR#*x}
xxyyzzz
> echo ${VAR##*x}
yyzzz

> VAR=/var/log/messages
> echo ${VAR##*/}
messages
```

- Remove a prefix from value of VAR
  - # = shortest match
  - ## = longest match

ADVANCED EXPANSION

${VAR%{%}*word*}

```
> VAR=/var/log/messages.txt
> echo ${VAR%.txt}.log
/var/log/messages.log

> echo ${VAR%%.*}
/var/log/messages
```

- Remove a suffix from value of VAR
  - % = shortest match
  - %% = longest match

ADVANCED EXPANSION

${VAR/{#|%}*pattern*/*word*}

```
> VAR=/var/log/messages.txt
> echo ${VAR/log?/}
/var/messages.txt
```

- Find the first occurrence of *pattern* in VAR and replace with *word*
  - The # and % anchor the search to the start and end of the value of VAR, respectively

ADVANCED EXPANSION

${VAR//*pattern*/*word*}

```
> VAR=/var/log/messages.txt
> echo ${VAR//a/^}
/v^r/log/mess^ges.txt
```

- Find all occurrences of *pattern* in VAR and replace with *word*

ARRAY VARIABLES

- Just as a variable can be declared integer-valued…
  - …declare MY_ARRAY as array-valued:

```
[frey@mills ~]$ declare -a MY_ARRAY

[frey@mills ~]$ MY_ARRAY[0]="This"
[frey@mills ~]$ MY_ARRAY[1]="is"
[frey@mills ~]$ MY_ARRAY[2]="Florida"
```

- Alternatively, just declare as a list of values:

```
[frey@mills ~]$ MY_ARRAY=("This" "is" "Florida")
```

ARRAY VARIABLES

- Access values in array by index:
  - ${name[index]}
    - index can be a static integer or a variable name

```
[frey@mills ~]$ echo ${MY_ARRAY[0]}
This

[frey@mills ~]$ i=2; echo ${MY_ARRAY[i]}
Florida
```

  - Values at all indices:

```
[frey@mills ~]$ echo ${MY_ARRAY[*]}
This is Florida
```

ARRAY VARIABLES

- Access values in array by index:
  - ${name[index]}
    - index can be a static integer or a variable name

```
[frey@mi
This

[frey@mi
Florida
```

*What might happen if I use @ rather than * here?*

  - V

```
[frey@mills ~]$ echo ${MY_ARRAY[*]}
This is Florida
```

ARRAY VARIABLES

- Delete values in array by index:
  - unset 'name[index]'

```
[frey@mills ~]$ unset 'MY_ARRAY[0]'
[frey@mills ~]$ echo ${MY_ARRAY[*]}
is Florida

[frey@mills ~]$ declare | grep ^MY_ARRAY=
MY_ARRAY=([1]="is" [2]="Florida")
[frey@mills ~]$ echo ${#MY_ARRAY[@]}
2
```

- Delete entire array:

```
[frey@mills ~]$ unset MY_ARRAY
[frey@mills ~]$ declare | grep MY_ARRAY
[frey@mills ~]$
```

ARRAY VARIABLES

- Count values in array:
  - ${#name[@]}

```
[frey@mills ~]$ echo ${#MY_ARRAY[@]}
2
```

ARRAY VARIABLES

- Loop over array values:

```
[frey@mills ~]$ MY_ARRAY=(This is Florida)
[frey@mills ~]$ i=0; for w in "${MY_ARRAY[@]}"; do
>   printf "arg #%d = %s\n" $i "$w"
>   i=$((i+1))
> done
arg #0 = This
arg #1 = is
arg #2 = Florida
```

ARRAY VARIABLES

- Loop over array values:

```
[frey@mills ~]$ MY_ARRAY=(This is Florida)
[frey@mills ~]$ for (( i=0 ; i < ${#MY_ARRAY[@]} ; i++ )); do
>   printf "arg #%d = %s\n" $i "${MY_ARRAY[i]}"
> done
arg #0 = This
arg #1 = is
arg #2 = Florida
```

ARRAY VARIABLES

- Loop over array values:

```
[frey@mills ~]$ MY_ARRAY=(This is Florida)
[frey@mills ~]$ for (( i=0 ; i < ${#MY_ARRAY[@]} ; i++ )); do
>   printf "arg #%d = %s\n" $i "${MY_ARRAY[i]}"
> done
arg #0 = This
arg #1 = is
arg #2 = Florida
```

The indices must be sequential for this method to work. A *sparse array* must be enumerated using a loop over words, not indices.

SUBROUTINES

- Encapsulate often-used command sequences in a sub-program

```
# If the single argument is a full SCSI disk (not a partition of a disk)
# print its id.
filter_and_print_scsi_disks () {
  if [[ $1 =~ ^scsi-([0-9a-f]+)$ ]]; then
    echo "SCSI disk found, id = ${BASH_REMATCH[1]}"
    return 0
  fi
  return 1
}

# Loop over disks by identifier:
for disk in $(ls -1 /dev/disk/by-id | sed 's/ ->.*$//'); do
  filter_and_print_scsi_disks "$disk"
  if [ $? -ne 0 ]; then echo "Not a full SCSI disk: $disk"; fi
done
```

SUBROUTINES

- Encapsulate often-used command sequences in a sub-program
  - Function arguments treated same as command-line arguments
  - Return value from function is an integer, treated akin to program return codes

SUBROUTINES

- Encapsulate often-used command sequences in a sub-program
  - Can be used in command substitutions, too

```
disks_by_id () {
  /bin/ls -1 /dev/disk/by-id | sed 's/ ->.*$//'
}

# Loop over disks by identifier:
for disk in $(disks_by_id); do
  echo "$disk"
done
```

CASE STATEMENTS

- Many-valued branch table
  - More concise, clear than a lengthy if…elif…else…fi

```
case "$1" in

  start|begin)
    # Commands to startup the daemon
        :
    ;;

  stop|end)
    # Commands to stop the daemon
        :
    ;;

  *)
    # For any other value:
    echo "Unknown option: $1"
    exit 1

esac
```

CASE STATEMENTS

- Branch labels can be simple patterns

```
case "$1" in

  [a-m]*)
    echo "$1 starts in the first half of the alphabet"
    ;;

  [n-z]*)
    echo "$1 starts in the second half of the alphabet"
    ;;

  *)
    # For any other value:
    echo "$1 is not a word"
    exit 1

esac
```

AND/OR LISTS

- Recall that return code 0 from program = success
  - *and list* chains a sequence of commands using &&
    - first to fail breaks out of the list

```
$(ls -1 /dev/sda &> /dev/null) && $(ls -1 /dev/sda5 &> /dev/null) && echo "OK"
```

AND/OR LISTS

- Recall that return code 0 from program = success
  - *and list* chains a sequence of commands using &&
    - first to fail breaks out of the list

```
$(ls -1 /dev/sda &> /dev/null) && $(ls -1 /dev/sda5 &> /dev/null) && echo "OK"
```

What might this output redirection operator do?

AND/OR LISTS

- Recall that return code 0 from program = success
  - *and list* chains a sequence of commands using &&
    - first to fail breaks out of the list
  - *or list* chains a sequence of commands using | |
    - first to succeed breaks out of the list

```
[ ! -f "$file" ] || (rm -f $file; echo "File \"$file\" deleted.")
```

SUBSHELLS

```
(echo "Deleting file \"$file\"."; rm $file 2> /dev/null)
```

- Surround one or more commands (separated by semicolons) with parentheses
  - Start a new *subshell*
  - Execute the commands
  - Return code of final command = return code of subshell

EXERCISES

QUESTION 1

```
> cat a_file
Hello, friend!

Greetings.

> tee 0> a_file
```

- Would you expect the command at the left to work? Why or why not?

QUESTION 1

```
> cat a_file
Hello, friend!

Greetings.

> tee 0> a_file
tee: read: Bad file descriptor
> cat a_file
>
```

- Would you expect the command at the left to work? Why or why not?
  - No. File descriptor zero (0) is an input source (read mode). The ">" opens the target file in write mode to accept output. Since it was ">" the file was truncated, too, destroying the data in it in the process!

QUESTION 2

```
> dd if=/dev/zero of=a_file \
     bs=10 count=12
```

- What does the command at the left do? Feel free to use *man dd* on a Unix/Linux system to research the "dd" command.

QUESTION 2

```
> dd if=/dev/zero of=a_file \
     bs=10 count=12
12+0 records in
12+0 records out
120 bytes transferred in 0.000067 …
> hexdump a_file
0000000 00 00 00 00 00 00 00 00 00 …
*
0000070 00 00 00 00 00 00 00 00
0000078
>
```

- What does the command at the left do?
  - It reads 10 bytes from /dev/zero 12 times and writes what was read to a_file. This produces a file containing 120 bytes, all zero.

QUESTION 2

```
> dd if=/dev/zero of=a_file \
     bs=10 count=12
12+0 records in
12+0 records out
120 bytes transferred in 0.000067 …
> hexdump a_file
0000000 00 00 00 00 00 00 00 00 00 …
*
0000070 00 00 00 00 00 00 00 00
0000078
>
```

Shown in hexadecimal (base 16), so 0x78 = (7*16)+8 = 120

- What does the command at the left do?
  - It reads 10 bytes from /dev/zero 12 times and writes what was read to …is produces a …ining 120 bytes,

QUESTION 2

```
> dd if=/dev/zero of=a_file \
     bs=10 count=12
12+0 records in
12+0 records out
120 bytes transferred in 0.000067 …
> hexdump a_file
0000000 00 00 00 00 00 00 00 00 00 …
*
0000070 00 00 00 00 00 00 00 00
0000078
> dd if=/dev/random of=a_file \
     bs=10 count=12
```

- What does the command at the left do?
- What if I used /dev/random instead?

QUESTION 2

```
> dd if=/dev/zero of=a_file \
     bs=10 count=12
12+0 records in
12+0 records out
120 bytes transferred in 0.000067 …
> hexdump a_file
0000000 00 00 00 00 00 00 00 00 00 …
*
0000070 00 00 00 00 00 00 00 00
0000078
> dd if=/dev/random of=a_file \
     bs=10 count=12
12+0 records in
12+0 records out
120 bytes transferred in 0.000108 …
> hexdump a_file
0000000 b7 ba cc 63 4d 71 28 c6 3f …
0000010 31 c3 a8 fd c1 a7 fb fb 18 …
0000020 78 c0 d8 b8 b8 ab b0 34 65 …
0000030 8c fd d2 7b d2 ff d8 f6 9b …
0000040 37 d0 86 82 b6 54 f1 05 89 …
0000050 d4 68 d6 1d 90 93 79 49 d0 …
   :
```

- What does the command at the left do?
- What if I used /dev/random instead?
  - The file will contain 120 pseudo-random bytes.

QUESTION 3

```
> echo ${VAR:+${VAR:=xyz}}
```

- When would this command produce "xyz" as its output?

QUESTION 3

```
> unset VAR
> echo ${VAR:+${VAR:=xyz}}

> VAR=abc
> echo ${VAR:+${VAR:=xyz}}
abc
> VAR=xyz
> echo ${VAR:+${VAR:=xyz}}
xyz
>
```

- When this command produce "xyz" as its output?
  - The ":+" syntax evaluates the second expression if VAR has a value. Since VAR has a value, the second expression simply returns the value of VAR.
  - Setting VAR=xyz is the only way this produces that text.

QUESTION 4

```
#!/bin/bash

debug () {
  echo "DEBUG: $@ ($BASH_SOURCE:${BASH_LINENO[0]})"
}

debug Starting execution...
echo "Here we are!"
if [ 1 == 1 ]; then
  debug One is always equal to itself, right?
fi
```

- Consider the script at the left.
  - What is "debug"?
  - What kind of variable is BASH_LINENO?
  - What does this script produce when executed?

QUESTION 4

```
#!/bin/bash

debug () {
  echo "DEBUG: $@ ($BASH_SOURCE:${BASH_LINENO[0]})"
}

debug Starting execution...
echo "Here we are!"
if [ 1 == 1 ]; then
  debug One is always equal to itself, right?
fi
```

- Consider the script at the left.
  - What is "debug"?
    - A subroutine

QUESTION 4

```
#!/bin/bash

debug () {
  echo "DEBUG: $@ ($BASH_SOURCE:${BASH_LINENO[0]})"
}

debug Starting execution...
echo "Here we are!"
if [ 1 == 1 ]; then
  debug One is always equal to itself, right?
fi
```

- Consider the script at the left.
  - What kind of variable is BASH_LINENO?
    - An array

QUESTION 4

```
#!/bin/bash

debug () {
  echo "DEBUG: $@ ($BASH_SOURCE:${BASH_LINENO[0]})"
}

debug Starting execution...
echo "Here we are!"
if [ 1 == 1 ]; then
  debug One is always equal to itself, right?
fi
```

- Consider the script at the left.
  - What does this script produce when executed?

```
> ./question_3
DEBUG: Starting execution... (./question_3:7)
Here we are!
DEBUG: One is always equal to itself, right? (./question_3:10)
>
```

QUESTION 4

```
VALUE=000014

while [[ $VALUE =~ ^0 ]]; do
  VALUE=${VALUE#0}
done
echo $VALUE
```

- What is the final output of the code at the left?

QUESTION 4

```
VALUE=000014

while [[ $VALUE =~ ^0 ]]; do
  VALUE=${VALUE#0}
done
echo $VALUE
```

- What is the final output of the code at the left?
  - 14. The loop uses a regular expression that is true so long as VALUE begins with a zero character, and drops the leading "0" each pass.

QUESTION 4

```
VALUE=000014

while [[ $VALUE =~ ^0 ]]; do
  VALUE=${VALUE#0}
done
echo $VALUE
```

- What is the final output of the code at the left?
- How might this be better implemented using the regular expression capabilities of BASH?

QUESTION 4

```
VALUE=000014
if [[ $VALUE =~ ^0+(.+)$ ]]; then
  VALUE=${BASH_REMATCH[1]}
fi
echo $VALUE
```

- What is the final output of the code at the left?
- How might this be better implemented using the regular expression capabilities of BASH?
  - A single regex operation; exploit captured parenthesized piece

QUESTION 4

```
VALUE=000014
if [[ $VALUE =~ ^0+(.+)$ ]]; then
  VALUE=${BASH_REMATCH[1]}
fi
echo $VALUE
```

- What is the final output of the code at the left?
- Could this be implemented more simply another way?

QUESTION 4

```
shopt -s extglob

VALUE=000014
VALUE=${VALUE##*(0)}
echo $VALUE
```

- What is the final output of the code at the left?
- Could this be implemented more simply another way?
  - *Extended globbing* allows for more complex patterns; in this case, zero-or-more repetitions of the character "0"