CS 288 Intensive Programming in Linux

Professor Ding, Xiaoning

The shell of Linux

- Different linux shells: Bourne shell (sh), C shell (csh), Korn shell (ksh), TC shell (tcsh), Bourne Again shell (bash).
- Bash: the most popular Linux shell
  - It is a command line interface. We used it to type in and run commends.
  - It is a scripting language. It interprets and runs scripts. We will write bash scripts.
- Shell scripting uses the shell's facilities and existing software tools as building blocks to automate a lot of tasks.
  - Shell facilities: if, for loops, arrays, some built-in commands in shell (e.g., echo).
  - Existing software tools: grep, tr, uniq, …., any other executable files (binary and scripts).
  - Do not need to type in a lot of commands repeatedly.
  - Do not need to build programs from scratch (e.g., instructions).

The first bash program

- Create a script and save the script
  - The first line (Shebang) tells Linux to use the bash interpreter to run this script.
  - Note # also starts the comments. But first line is special.
- make the file executable using `chmod`.
- Run the script.
- Revise the script if it does not run correctly.

```
$vi hello.sh

 #!/bin/bash
 echo hello

$chmod 700 hello.sh
$./hello.sh
  hello

$vi hello.sh

 #!/bin/bash
 echo Hello
```

Including multiple commands in a script

```
$ mkdir trash
$ mv * trash
```

Using commands to create a directory and copy all files into that directory before removing them.

```
$ vi trash.sh

  #!/bin/bash
  mkdir trash
  mv * trash

$ ./trash.sh
```

Instead of having to type all the commands interactively on the shell, write a script

Bash scripts view all the data as texts/strings.

- When a text contains space, tab, newline, the text must be enclosed in either single or double quotes.

```
$ cat my file1.txt   #print out files "my" and "file1.txt"?
$ cat "my file1.txt" #print out file "my file1.txt"
```

- When a text contains special characters, to be safe, the text must be enclosed in either single or double quotes.
  – The parts with special characters may be translated and replaced (see "expansions"), and new text may contain space/tab/newline.

Variables

- Variable values are always stored as strings
  - Introduce later: How to convert variables to numbers for calculations?
- No need to declare a variable
  - assigning a value to a variable creates it.
- Value extracted using $
  - Use { } when necessary

```
$ cat variable.sh
#!/bin/bash
STR="Hello World!"
echo $STR
STR2=Hello
echo $STR2
echo ${STR}2
$ ./variable.sh
Hello World!
Hello
Hello World!2
```

Single and double quotes

When assigning character data containing spaces or special characters, the data must be enclosed in either single or double quotes.

Using double quotes to show a string of characters will allow any variables in the quotes to be resolved.

```
#!/bin/bash
var="test string"
newvar="Value of var is $var"
echo $newvar
```

Output: Value of var is test string

Using single quotes to show a string will not allow variable resolution.

```
#!/bin/bash
var='test string'
newvar='Value of var is $var'
echo $newvar
```

Output: Value of var is $var

Single and double quotes

- Quotes marking the beginning and end of a string are not saved in variables

```
#!/bin/bash
var="test string"
#get the first character, will introduce later
echo ${var:0:1} # echo prints letter t not quote
var="\"test string\"" #escape quotes to include them
echo ${var:0:1} # echo prints double quote
```

- Apply quotes properly when the string in a variable is retrieved and there exits space character(s) in the string.
  - Without quotes, space characters break one string into multiple strings.

```
#!/bin/bash
var="my file.txt"  #a space character in file name
cat $var           #cannot find the file
                   #cat: my: No such file or directory
                   #cat: file.txt: No such file or directory
cat "$var"         #print file content correctly
```

Scope of a variable

By default, all variables are global, even if declared inside a function.

- Can be accessed from anywhere in the script regardless of the scope.
- *Inaccessible* from outside of the script
- *Inaccessible* in other scripts run by the script defining the variable

```
$ cat a.sh
#!/bin/bash
a=hello
echo $a

$ ./a.sh
hello
$ echo $a

$
```

nothing is printed out

What if we want to make *b.sh* print out *"hello"*

```
$ cat a.sh
#!/bin/bash
a=hello
./b.sh

$ cat ./b.sh
#!/bin/bash
echo $a

$ ./a.sh

$
```

nothing is printed out

Environment variables and export command

```
$ cat a.sh
#!/bin/bash
export a=hello
./b.sh

$ cat ./b.sh
#!/bin/bash
echo $a

$ ./a.sh
hello
$
```

The ***export*** command makes a variable an **environment variable**, so it will be accessible from "children" scripts.

If a “child” script modifies an environment variable, it will NOT modify the parent’s original value.

```
$ cat ./a.sh
#!/bin/bash
export a=hello
./b.sh
echo $a

$ cat ./b.sh
#!/bin/bash
a=bye

$ ./a.sh
hello
$
```

Some common environment variables

- Created by the system for saving some system settings
- Can be found with the ***env*** command.
- Accessible in command line interface and any shell scripts.

```
$ echo $SHELL
/bin/bash
$ echo $PATH
/usr/X11R6/bin:/usr/local/bin:/bin:/usr/bin
$ cat a.sh
#!/bin/bash
echo $HOME
$ ./a.sh
/home/fall2020/tom
```

Some common environment variables

- ?: **exit status of previous command**
- LOGNAME, USER: contains the user name
- RANDOM: random number generator
- SECONDS: seconds from the beginning of the execution
- PS1: sequence of characters shown before the prompt

\t	hour	\d	date	\w	current directory
\W	last part of the current directory	\u	user name	\$	prompt character

Example:

```
$ PS1='hi \u *$'
hi userid*$ _
```

Read command

The read command allows you to prompt for input and store it in a variable.

```
#!/bin/bash
echo -n "Enter pathname of file to backup: "
read file_pathname
cp $file_pathname /home/tom/backup/
```

The script reads a pathname into variable ***file_pathname***, and copies the corresponding file into the backup directory.

Expansions: a few ways to operate texts

- Bash may perform a few types of expansions to commands before executing them.
- Replace special expressions with texts
  - variable expansion
  - brace expansion
  - tilde expansion
  - command substitution
  - arithmetic expansion
  - filename expansion

Variable expansion

${var} : string saved in var

${#var} gives the string length

${var:position} extracts sub-string from $string at $position

${var:position:length} extracts a sub-string of $length from $position

```
$ st=0123456789
$ echo ${#st}
     10
$ echo ${st:6}
     6789
$ echo ${st:6:2}
     67
```

Brace expansion and tilde expansion

Brace expansion expands a sequence expression or a comma separated list of items inside curly braces "{}"

- Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result.
- "${" for variable expansion is not considered eligible for brace expansion

```
$ echo  a{d,c,b}e
   ade ace abe
$ echo  a{0..3}b
   a0b a1b a2b a3b
$ mkdir home_{tom,berry, jim}
$ ls
home_berry  home_jim  home_tom
```

Tilt expansion replaces an unquoted tilde character "~" at the beginning of a word with pathname of home directory

~   : home directory of current user ($HOME)
~/foo : foo subdirectory under the home
~fred/foo : the subdirectory foo of the home
                  directory of the user fred

Command substitution: saving the output of a command into a variable

```
$ LIST=`ls`
$ echo $LIST
hello.sh read.sh

$ PS1="`pwd`>"
/home/userid/work> _
```

command substitution **using backquotes** : `command`

(use backquote "`" , not single quote "'").

Command substitution using $ and (): $(command)

```
$ LIST=$(ls)
$ echo $LIST
hello.sh read.sh

$ rm $( find / -name "*.tmp" )

$ cat > backup.sh
#!/bin/bash
BCKUP=/home/$USER/backup-$(date +%F).tgz
tar -czf $BCKUP $HOME
```

Evaluating arithmetic expressions

Translate a string into a numerical expression

- Command substitute and expr: ` expr expression` or $(expr expression)
  - e.g., `` z=`expr $z + 3` ``
  - Read manual of command expr
- double parentheses: $((expression))

Arithmetic expansion

```
$ echo "$((123+20))“
143
$ echo "$((123*$VALORE))"
$ echo "$((123*VALORE))"
```

- The let statement: let var=expression

```
 $ X=2; let X=10+X*7
 $ echo $X
 24
```

Arithmetic operators: +, -, /, *, %

```
$ cat arithmetic.sh
#!/bin/bash
echo -n "Enter the first number: "; read x
echo -n "Enter the second number: "; read y
add=$(($x + $y)); sub=$(($x - $y))
mul=$(($x * $y)); div=$(($x / $y))
mod=$(($x % $y));
echo "Sum: $add"
echo "Difference: $sub"
echo "Product: $mul"
echo "Quotient: $div"
echo "Remainder: $mod"
```

filename expansion

Bash scans each word for the characters ‘*****’, ‘**?**’, and ‘**[**’. If one of these characters appears, then the word is regarded as a pattern, and **replaced with an alphabetically sorted list of filenames matching the pattern**.

- ***** Matches any string, including the null string.
- **?** Matches any single character.
- **[…]** Matches any one of the enclosed characters.

```
$ ls *.pdf
$ ls fig?.pdf
$ ls fig[0-9].pdf
$ ls fig_[abc].pdf
```

```
$ mkdir home{1..3}
$ mkdir home{1,2}{a..c}
$ echo home*
$ echo home*
home1 home1a home1b home1c home2 home2a home2b home2c home3
$ echo home[12345]
home1 home2 home3
$ echo home?[bc]
home1b home1c home2b home2c
```

Spaces and word Splitting

The shell scans the results of variable expansion, command substitution, and arithmetic expansion for word splitting.

- Results from filename expansion are not spitted
- Usually happens when the results are used in command lines, not in assignments
- double quotes prevent world splitting

```
$ echo "Hello      World"
"Hello      World"
$ a="Hello      World"
$ echo ${a}
Hello World
$ echo ${a#}
16
$ echo "${a}"
Hello      World
$b=$a
$ echo ${b#}
16
```

rule of thumb: double-quote every expansion except filename expansion

Conditional statements

```
if COMMANDS
then
    statements
elif COMMANDS
then
    statements
else
    statements
fi
```

```
if COMMANDS; then
    statements
elif COMMANDS; then
    statements
else
    statements
fi
```

```
if COMMANDS; then statements; elif COMMANDS; then statements; else statements; fi
```

- elif (else if) and else sections are optional
- Conditions are exit code ($?) of COMMAND

Conditional statements

```
if [ expression ]; then
    statements
elif [ expression ]; then
    statements
else
    statements
fi
```

- [is a command usually used in if
  - [is another implementation of the traditional test command.
  - [or test is a standard POSIX utility.
  - Implemented in all POSIX shells.
- An expression can compare numbers, strings, check files, combine multiple conditions…
- Put spaces before and after each expression, and around the operators in each expression.

Comparing numbers

-eq compare if two numbers are equal

-ge compare if one number is greater than or equal to a number

-le compare if one number is less than or equal to a number

-ne compare if two numbers are not equal

-gt compare if one number is greater than another number

-lt compare if one number is less than another number

- Examples:

[n1 -eq n2] true if n1 same as n2, else false

[n1 -ge n2] true if n1greater then or equal to n2, else false

[n1 -le n2] true if n1 less then or equal to n2, else false

[n1 -ne n2] true if n1 is not same as n2, else false

[n1 -gt n2] true if n1 greater then n2, else false

[n1 -lt n2] true if n1 less then n2, else false

Examples

```
$ cat number.sh
#!/bin/bash
echo -n "Enter a number 1<x<10:"
read num
if [ $num -lt 10 ]; then
  if [ $num -gt 1 ]; then
    echo "$num*$num=$(($num*$num))"
  else
    echo "Wrong number!"
  fi
else
  echo "Wrong number!"
fi
```

Comparing strings

= compare if two strings are equal

!= compare if two strings are not equal

-n evaluate if string length is greater than zero

-z evaluate if string length is equal to zero

- Examples:

[s1 = s2]	true if s1 same as s2, else false
[s1 != s2]	true if s1 not same as s2, else false
[s1]	true if s1 is not empty, else false
[-n s1]	true if s1 has a length greater then 0, else false
[-z s2]	true if s2 has a length of 0, otherwise false

```
$ cat user.sh
#!/bin/bash
echo -n "Enter your login
   name: "
read name
if [ "$name" = "$USER" ];
then
   echo "Hello, $name."
else
   echo "You are not $USER"
fi
```

Checking files/directories

-e check if file/path name exists
-d check if path given is a directory
-f check if path given is a file
-r check if read permission is set for file or directory
-s check if a file has a length greater than 0
-w check if write permission is set for a file or directory
-x check if execute permission is set for a file or directory

- Examples:

[-d fname]	(true if fname is a directory, otherwise false)
[-f fname]	(true if fname is a file, otherwise false)
[-e fname]	(true if fname exists, otherwise false)
[-s fname]	(true if fname length is greater then 0, else false)
[-r fname]	(true if fname has the read permission, else false)
[-w fname]	(true if fname has the write permission, else false)
[-x fname]	(true if fname has the execute permission, else false)

```
#!/bin/bash
read fname
if [ -f $fname ]; then
  cp $fname .
  echo "Done."
else
  if [ -e $fname ]; then
    echo "Not a file."
  else
    echo "Not exist."
  fi
  exit 1
fi
```

Exercise

Write a shell script which:

- Allows user to type in a file name (e.g., ./myfile.txt)
- checks if the file exists
- if the file exists, make a copy of the file under the same directory. Append a “.bak” to the file name of the copy (e.g., ./myfile.txt.bak).
- If the file does not exist, print out “file does not exist.”

Logically operators: AND (-a, &&), OR (-o, ||), NOT (!)

```
#!/bin/bash
echo -n "Enter a number 1<x<10:"
read num
if [ $num -lt 10 ]; then
  if [ $num -gt 1 ]; then
    echo "$num*$num=$(($num*$num))"
  else
    echo "Wrong number!"
  fi
else
  echo "Wrong number!"
fi
```

```
#!/bin/bash
echo -n "Enter a number 1<x<10:"
read num
if [ $num -lt 10 -a $num -gt 1 ]; then
    echo "$num*$num=$(($num*$num))"
else
  echo "Wrong number!"
fi
```

Pay attention to the forms when combining conditions

```
if [ condition1 ] && [ condition2 ]
if [ condition1 -a condition2 ]
if [ condition1 ] || [ condition2 ]
if [ condition1 -o condition2 ]
```

```
#!/bin/bash
echo -n "Enter a number 1<x<10:"
read num
if [ $num -gt 1 ] && [ num -lt 10 ];
  then
  echo "$num*$num=$(($num*$num))"
else
  echo "Wrong number!"
fi
```

Case statement

```
case var in
val1)
    statements;;
val2)
    statements;;
*)
    statements;;
esac
```

- Execute statements based on specific values.
- each set of statements must be ended by a pair of semicolons;
- a *) is used to accept any value not matched with list of values

```
$ cat case.sh
#!/bin/bash
echo -n "Enter a number 1 < x < 10: "
read x
case $x in
          1) echo "Value of x is 1.";;
          2) echo "Value of x is 2.";;
          3) echo "Value of x is 3.";;
          4) echo "Value of x is 4.";;
          5) echo "Value of x is 5.";;
          6) echo "Value of x is 6.";;
          7) echo "Value of x is 7.";;
          8) echo "Value of x is 8.";;
          9) echo "Value of x is 9.";;
          0 | 10) echo "wrong number.";;
          *) echo "Unrecognized value.";;
esac
```

for loop

```
for VARIABLE in PARAM1 PARAM2 PARAM3
do
  statements
done
```

- for loop executes for each param in the list.
- The VARIABLE is initialized with a param value which can be accessed in inside the for loop scope
- Param can be any number, string etc.

```
#!/bin/bash
let sum=0
for num in 1 2 3 4 5
do
   let "sum = $sum + $num"
done
echo $sum
```

```bash
#!/bin/bash
for x in paper pencil pen
do
    echo "The value of variable x is: $x"
    sleep 1
done

for x in paper "a pencil"  "two pens"
do
    echo "The value of variable x is: $x"
    sleep 1
done
```

Example: Changes all filenames to lowercase

```bash
#!/bin/bash
# for all files in a directory.
for filename in `ls ./*`
do
  # filename in lowercase.
  n=`echo $filename | tr A-Z a-z`
  # Rename only files not already lowercase.
  if [ "$filename" != "$n" ]; then
             mv $filename $n
      fi
done
exit 0
```

Using range in a for loop

```
#!/bin/bash
for value in {1..5}
do
    echo $value
done
for value in {10..0..2}
do
    echo $value
done
```

Range: *{start..end}*, or *{start..end..step}*

- *Start* and *end* determine the direction (counts up/down)
- *Step* determines the increment (no need to be negative when counting down).
- Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result.
  - "${" for variable expansion is not considered eligible for brace expansion
  - Use **seq** instead.

seq FIRST INCREMENT LAST

```
#!/bin/bash
begin=1
end=5
for value in {${begin}..${end}}
do
    echo $value
done
```

Invalid

```
#!/bin/bash
begin=1
end=5
for value in `seq ${begin} ${end}`
do
    echo $value
done
```

Using range in a for loop

```
#!/bin/bash
for value in {1..5}
do
    echo $value
done
for value in {10..0..2}
do
    echo $value
done
```

Range: *{start..end}*, or *{start..end..step}*

- *Start* and *end* determine the direction (counts up/down)
- *Step* determines the increment (no need to be negative when counting down).
- Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result.
- "${" for variable expansion is not considered eligible for brace expansion

for loop in C style

First, the arithmetic expression EXPR1 is evaluated.

EXPR2 is then evaluated repeatedly until it evaluates to 0.

Each time EXPR2 is evaluates to a non-zero value, statements are executed and EXPR3 is evaluated.

```
for (( EXPR1; EXPR2; EXPR3 ));
do
   statements
done
```

```
$ cat ./mysum.sh
#!/bin/bash
echo -n "Enter a number: "
read x
sum=0
for ((i=1;i<=x;i=i+1)) ; do
   sum=$(($sum+$i))
done
echo "Sum of 1...$x is: $sum"
```

While structure

Execute a set of commands while a specified condition is true.

- The loop terminates as soon as the condition becomes false.
- If condition never becomes false, loop will never exit.

```
while [ some_test ]
do
  statements
done
```

```
$ cat while.sh
#!/bin/bash
echo -n "Enter a number: "
read x
sum=0; i=1
while [ $i -le $x ]; do
  let "sum = $sum + $i"
  let "i = $i + 1"
done
echo "sum of 1...$x is: $sum"
```

Menu

```
#!/bin/bash
clear ; loop=y
while [ "$loop" = y ] ;
do
  echo "Menu";  echo "===="
  echo "D: print the date"
  echo "W: print the users who are currently log on."
  echo "P: print the working directory"
  echo "Q: quit.";   echo
  read choice
  case $choice in
    D | d) date ;;
    W | w) who ;;
    P | p) pwd ;;
    Q | q) loop=n ;;
    *) echo "Illegal choice." ;;
  esac
  echo
done
```

Until structure: loops until the condition is true

```
until [ some_test ]
do
  statements
done
```

```
$ cat countdown.sh
#!/bin/bash
echo "Enter a number: "
read x
echo "Count down"
until [ "$x" -le 0 ]; do
   echo $x
   x=$(($x -1))
   sleep 1
done
```

Continue: skip the remaining part in current iteration and jump to the next iteration

```
$ cat continue.sh
#!/bin/bash
echo "Print numbers 1 to 20 (but not 3 and 11)"
a=0
while [ $a -le 19 ]; do
    a=$(($a+1))
    if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]; then
     continue
    fi
    echo -n "$a "
done
```

Break terminates the loop

```
$ cat break.sh
#!/bin/bash
echo "Print numbers 1 through 20, but nothing after 12"
a=0
while [ $a -le 19 ];  do
      a=$(($a+1))
      if [ "$a" -gt 12 ]; then
             break
      fi
      echo -n "$a "
done
echo
```

Using arrays

- Bash does not offer lists, tuples, etc. **Just arrays**.
- bash has two types of arrays: one-dimensional indexed arrays and associative arrays
- An array is a variable containing multiple values.
- No maximum limit to the size of an array.
- No requirement that member variables be indexed or assigned contiguously

Index arrays

- Arrays are **zero-based**: the first element is indexed with the number 0.
- Creating an array
  - First way:

```
#3 elements
pet=("a dog" "a cat" fish)
#2 elements
pet=([2]=fish [0]="a dog")
```

  - Second way:

```
pet[0]="a dog"
pet[1]="a cat"
pet[2]=fish
```

- Third way:

```
#brace expansion
pet=(a{1..3})
#(a1 a2 a3)
#filename expansion
files=(./*)
```

Using index arrays

- To extract a value: ${arrayname[i]}

  $ echo ${pet[0]}

  a dog

```
pet=("a dog" "a cat" fish)
echo $pet        # a dog
echo $pet[1]     # a dog[1]
echo ${pet[1]}   # a cat
```

- extract all the elements:

  ${arrayname[*]}, ${arrayname[@]}
- extract the count of the elements: ${#arrayname[@]}
- Extract all the indices that have been assigned: ${!arrayname[@]}
- extracts sub-array at $position: ${arrayname[@]:position}
- extracts $length elements from $position:

  ${arrayname[@]:position:length}
- Search and replace an element: ${arrayname[@]:OldText:NewText}
- Add new elements: arrayname+=(new_ele1 new_ele2)
- Delete an element: unset arrayname[index]

Associative arrays

- The index can be any arbitrary string.
- Creation: must be declared with ***typeset -A*** or ***declare –A***
- Individual element can be accessed using the index string.
- features of indexed arrays are available to associative arrays.

```
#!/bin/bash
declare -A shade
shade[apple]="dark red"
shade[banana]="bright yellow"
#add a new element
shade+=([grape]=purple)
for i in apple banana grape
do
  echo ${shade[$i]}
done
for i in ${!shade[@]}; do
  echo $i ${shade[$i]}
done
#remove an element
unset shade[apple]
```

Example: Picking a random poker card (random suit & random rank)

```
#!/bin/bash
Suits="Clubs Diamonds Hearts Spades"
Ranks="2 3 4 5 6 7 8 9 10 Jack Queen King Ace"

# Read into array variable.
suit=($Suits)
rank=($Ranks)

# Count how many elements.
num_suits=${#suit[*]}
num_ranks=${#rank[*]}
echo -n "${rank[$(($RANDOM%num_ranks))]} of "
echo ${suit[$(($RANDOM%num_suits))]}
```

${arrayname[*]} and ${arrayname[@]}

- ${arrayname[*]} and ${arrayname[@]} are all the words in all the elements (as if elements are merged and divided into words)
  - `pet=("a dog" "a cat" fish)`
  - ${pet[*]} and ${pet[@]} get the contents in all elements and put them together: `a dog a cat fish`
- **"**${arrayname[*]}**"** : a single string containing all the words from all the elements (all words in the same pair of quotes)
  - "${pet[*]} " gets the contents in all elements, puts them together and inside double quotes: `"a dog a cat fish"`
- **"**${arrayname[@]}**"** : a string for each element (each element has a pair of quotes)
  - For each element, "${pet[@]} " gets its content and puts it inside double quotes : `"a dog" "a cat" "fish"`

Using array in a loop

```
$ cat arrayele.sh
#!/bin/bash
array=(one two three four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in ${array[*]}
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

```
$ cat arrayele.sh
#!/bin/bash
array=(one "two three" four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in ${array[*]}
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

Using array in a loop

```
$ cat arrayele.sh
#!/bin/bash
array=(one two three four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in ${array[@]}
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

```
$ cat arrayele.sh
#!/bin/bash
array=(one "two three" four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in ${array[@]}
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

Using array in a loop

```
$ cat arrayele.sh
#!/bin/bash
array=(one two three four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in "${array[*]}"
do
    echo $item
done
$ ./arrayele.sh
One two three four
```

```
$ cat arrayele.sh
#!/bin/bash
array=(one "two three" four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in "${array[*]}"
do
    echo $item
done
$ ./arrayele.sh
one two three four
```

Using array in a loop

```
$ cat arrayele.sh
#!/bin/bash
array=(one two three four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in "${array[@]}"
do
    echo $item
done
$ ./arrayele.sh
one
two
three
four
```

```
$ cat arrayele.sh
#!/bin/bash
array=(one "two three" four)
echo "Array size:${#array[*]}"
echo "Array items:"
for item in "${array[@]}"
do
    echo $item
done
$ ./arrayele.sh
one
two three
four
```

Example: Changes all filenames to lowercase

```
#!/bin/bash
#filename expansion into an array
files=(*)
for filename in "${files[@]}"
do
  # filename in lowercase.
  n=`echo $filename | tr A-Z a-z`
  # Rename only files not already lowercase.
  if [ "$filename" != "$n" ]; then
             mv $filename $n
      fi
done
exit 0
```

Shell parameters

- Positional parameters are assigned from arguments when a script is invoked.
- N-th positional parameter is ${N} or $N when N is single digit.
  - $1 : first command line argument
  - $0 : the name of the script
- Other special parameters
  - $#   the number of parameters passed
  - $*   all positional parameters except $0
  - $@   all positional parameters except $0

```
$ cat sparameters.sh
#!/bin/bash
echo "$#; $0; $1; $2; $*; $@"
$ sparameters.sh arg1 "arg #2"
2; ./sparameters.sh; arg1; arg #2; arg1 arg #2; arg1 arg #2
```

Example: Trash

```
$ cat trash.sh
#!/bin/bash
if [ $# -eq 1 ]; then
  if [ ! -d "$HOME/trash" ]; then
         mkdir "$HOME/trash"
  fi
  mv $1 "$HOME/trash"
else
  echo "Use: $0 filename"
  exit 1
fi
```

Difference between $* and $@

```
$ cat args.sh
#!/bin/bash
echo "Arg list as a single string"
index=1;
for arg in "$*" ; do
        echo "Arg $index = $arg"
        let "index+=1"
done

echo; index=1;
echo "Arg list as separate strings"
for arg in "$@" ; do
        echo "Arg $index = $arg"
        let "index+=1"
done
```

```
$ ./args.sh arg1 "arg2 arg3" arg4
Arg list as a single string
Arg 1 = arg1 arg2 arg3 arg4

Arg list as separate strings
Arg 1 = arg1
Arg 2 = arg2 arg3
Arg 3 = arg4
```

Indirection with !

What does the following script print out?

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

What is $i?
Is it $0, $1, …?

```
#!/bin/bash
for
((i=0;i<=$#;i++)); do
    echo ${!i}
done
```

Iterate arguments

When the list part in a for loop is left off, var is set to each argument ($1, $2, $3,…)

```
$ cat for1.sh
#!/bin/bash
for x
do
     echo "The value of variable x is: $x"
     sleep 1
done
$ for1.sh arg1 arg2
The value of variable x is: arg1
The value of variable x is: arg2
```

Functions

- Functions are like mini-scripts. They can
  - accept parameters ($1, $2, …)
  - create variables only known within the function
  - return values to the calling shell (not caller).
- A function is called by its name

```
function name
{
  commands;
  return x;
}

function name()
{
  commands;
  return;
}
```

```
$ cat function.sh
#!/bin/bash
function check()
{
if [ -e "/home/$1" ]; then
  return 0
else
  return 1
fi
}
echo "Enter a file name:"
read x
if check $x
then
  echo "$x exists !"
else
  echo "$x not exists!"
fi.
```

Variables created in a function and local variables

- In contrast to C, a Bash variable declared inside a function is local ONLY IF declared as such.

```
        local var_name
```

- If not declared as local, variables are global by default.
- Before a function is called, all variables declared within the function are invisible outside the body of the function, not just those explicitly declared as local.

```
$ cat ./var_in_func.sh
#!/bin/bash
func ()
{
  local loc_var=23   # Declared as local variable.
  echo "\"loc_var\" in function = $loc_var"
  global_var=999
  echo "\"global_var\" in function = $global_var"
}

func
# $loc_var not visible globally.
echo "\"loc_var\" outside function = $loc_var"
# $global_var is visible globally.
echo "\"global_var\" outside function = $global_var"
$ ./var_in_func.sh
loc_var outside function =
global_var outside function = 999
```

```
$ cat ./var_in_func.sh
#!/bin/bash
func ()
{
global_var=37
}

# $global_var is not visible here. "func" not called,
echo "global_var = $global_var"
func
# $global_var has been set by function call.
echo "global_var = $global_var"
$ ./var_in_func.sh
global_var =
global_var = 37
```

Return a value from Bash functions

Using a global variable

```
#!/bin/bash
function F1()
{
  retval='Like programming'
}

retval='Hate programming'
echo $retval
F1
echo $retval
```

Using function command

```
#!/bin/bash
function F2()
{
  local retval='BASH Func'
  echo "$retval"
}

getval=$(F2)
echo $getval
```

Return a value from Bash functions using $?

```
#!/bin/bash -x

function factorial()
{
   if (( $1 < 2 ))
   then
     return 1
   else
     factorial $(( $1 - 1 ))
     result=$(( $1 * $? ))
     return ${result}
   fi
}

factorial $1
echo $?
```

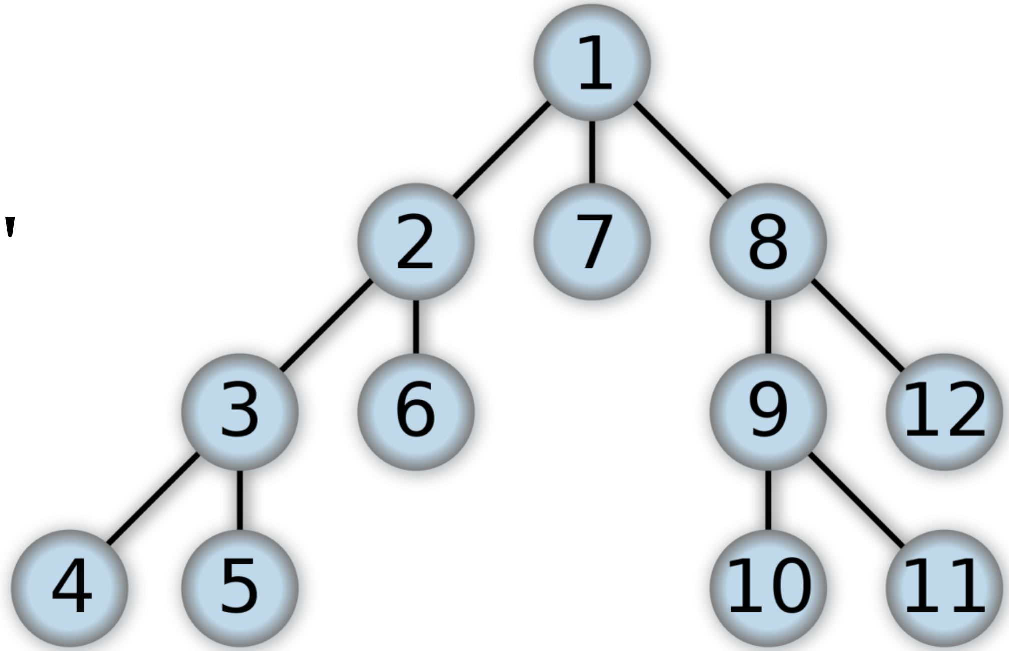
- Problem: $? must be an integer in the 0 - 255 range
- The code on the left works for 1, 2, ..., 5, but not 6.

Example: factorial of a number

```
#!/bin/bash

function factorial()
{
   if (( $1 < 2 ))
   then
     echo 1
   else
     echo $(( $1 * $(factorial $(( $1 - 1 ))) ))
   fi
}

factorial $1
```

Example: traverse a directory (**depth-first**)

```
#!/bin/bash
traverse() {
echo $1
entries=("$1"/*)
for entry in "${entries[@]}"
do
    traverse "$entry"
done
}

traverse "$1"
```



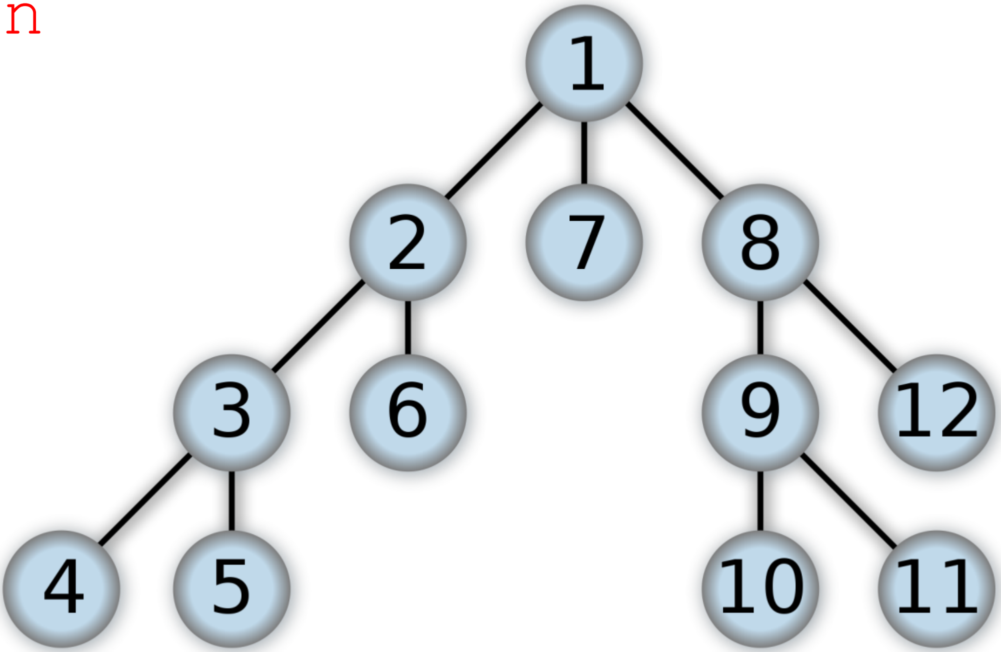
Can this code traverse correctly?

Example: traverse a directory (**depth-first**)

```
#!/bin/bash
traverse() {
echo $1
if [ ! -d "$1" ]; then
    return
fi
if [ `ls "$1" | wc -l` -eq 0 ]; then
   return
fi
local entries=("$1"/*)
local entry
for entry in "${entries[@]}"
do
    traverse "$entry"
done
}

traverse "$1"
```

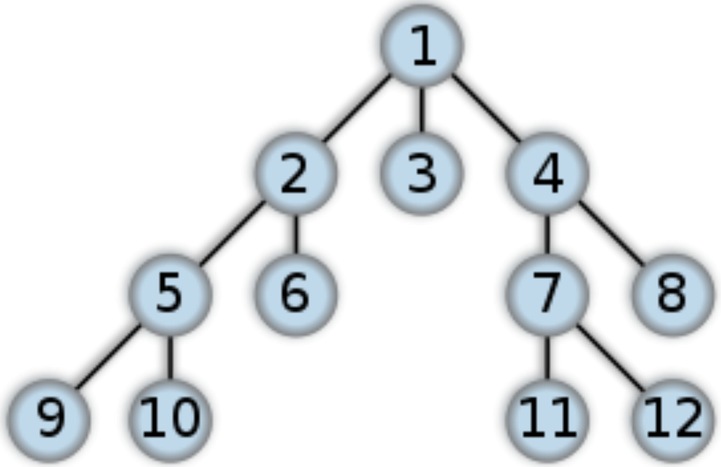
What if we remove the quotes?

What if we remove "local"?



Example: traverse a directory (**breadth-first**)

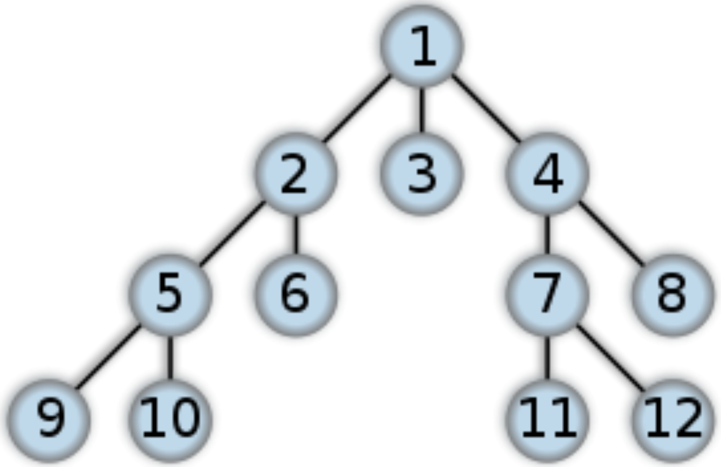
```
#!/bin/bash
function traverse(){
  if [ ${#queue[@]} -eq 0 ]; then return; fi
  echo ${queue[0]}
  if [ -d "${queue[0]}" ] && [ `ls "${queue[0]}" | wc -l` -ne 0 ]
  then
    entries=("${queue[0]}"/*)
    #merge two arrays
    queue=("${queue[@]}" "${entries[@]}")
  fi
  queue=("${queue[@]:1}") #remove elem #0
  traverse
}

queue[0]="$1"
traverse
```



Example: traverse a directory (**breadth-first**)

recursion ➔ loop

```
#!/bin/bash
queue[0]="$1"
while [ ${#queue[@]} -ne 0 ]; do
  echo ${queue[0]}
  if [ -d "${queue[0]}" ] && [ `ls "${queue[0]}" | wc -l` -ne 0 ]
  then
    entries=("${queue[0]}"/*)
    #merge two arrays
    queue=("${queue[@]}" "${entries[@]}")
  fi
  queue=("${queue[@]:1}") #remove elem #0
done
```



Debugging

Two debug options on the first script line:

#!/bin/bash -v or #!/bin/bash -x

-v : displays each line of the script as typed before execution

-x : displays each line of the script with variable substitution and before execution

```
$ cat for3.sh
#!/bin/bash -x
echo -n "Enter a number: "; read x
sum=0
for ((i=0;i<=x;i=i+1)); do
  sum=$(($sum + $i))
done
echo "the sum of 1...$x is: $sum"
```

```
$ ./for3.sh
+ echo -n 'Enter a number: '
Enter a number: + read x
2
+ sum=0
+ (( i=0 ))
+ (( i<=x ))
+ sum=0
+ (( i=i+1 ))
+ (( i<=x ))
+ sum=1
+ (( i=i+1 ))
+ (( i<=x ))
+ sum=3
+ (( i=i+1 ))
+ (( i<=x ))
+ echo 'the sum of 1...2 is: 3'
the sum of 1...2 is: 3
```

Programming or scripting?

- Programming languages are faster
  - source code is compiled into an executable. One time translation effort, and a lot of optimization during compilation.
  - script is not compiled into an executable. An interpreter reads, interprets, and executes the statements in a script. A lot of format conversions. Some inconvenience (e.g., lack of types and formats).
- Programming languages are usually more flexible and powerful: more facilities and various libraries.
- Scripts: fast development, easy to change/improve.
  - do not need to build programs from scratch (e.g., instructions).
- Common practice --- combining both: compiled parts for speed (e.g., building blocks), script parts for flexibility.