Shell Programming III

01418235

Chavalit Srisathapornphat

Outline

- Functions
- Argument validations
- Debugging scripts
- Variable substitutions (String operators)
  - Substring expansions
  - Pattern matching
  - Miscellaneous operators
- Arrays
- Input/Output
- Process handling

Functions (1)

- Writing functions
  - Functions must be placed at the beginning of the file
  - Passing arguments to a function is similar to passing arguments to a command line
    - Functions read them in a form of positional parameters
  - Function format

```
function_name()
{
    statement
    statement
    …
    return value
}
```

```
 1 #!/bin/bash
 2 # function01.sh
 3
 4 alist ()
 5 {
 6   ls -l $1 | more
 7 }
 8
 9 alist file1
10 alist file2
11 alist
```

Functions (2)

- Local variables
  - By default, all variables are global.
  - Modifying a variable in a function changes it in the whole script.
  - To create a local variables, use the `local` command
    - `local var=value`
    - `local varName`
  - `local` command can only be used within a function.

```
#!/bin/bash
# global d variable
d=/apache.jail

# User defined function
create_jail(){
    # d is only visible to this fucntion
    local d=$1
    echo "create_jail(): d is set to $d"
}

echo "Before calling create_jail  d is set to $d"
create_jail "/home/apache/jail"
echo "After calling create_jail d is set to $d"
```

Functions (3)

- Returning Values
  - return <status>
    - #status is 1-byte unsigned int (0-255)
  - If no status specified, 'return status' of previous command is returned
  - The return value is stored in exit status (`$?`)

```
 1 #!/bin/bash
 2 # function02.sh
 3 #
 4 add()
 5 {
 6   (( res = $1 + $2 ))
 7   return $res
 8 }
 9
10 add 4 5
11 echo "4 + 5 is " $?
12
13 add 42 96
14 echo "42 + 96 is " $?
```

Functions (4)

- Returning Values
  - Return a string/word

```
#!/bin/bash
# Variables
domain=$1
out=""

usage()
{
        echo "Usage: $0 domainname"
        exit 2
}

######################################################################
# Purpose: Converts a string to lower case
# Arguments:
#   $@ -> String to convert to lower case
######################################################################
function to_lower()
{
    local str="$@"
    local output
    output=$(tr '[A-Z]' '[a-z]'<<<"${str}")
    echo $output
}

[ $# -eq 0 ] && usage

# invoke the to_lower()
to_lower "This Is a TEST"

# invoke to_lower() and store its result to $out variable
out=$(to_lower ${domain})

# Display  back the result from $out
echo "Domain name : $out"
```

Argument Validation

- **$#** could be used to validate input
  - Fixed number of arguments
  - Minimum number of arguments
  - Numeric arguments
  - File type validation

Fixed Number Of Arguments

```
 1 #!/bin/sh
 2 # validate01.sh
 3 #
 4 if (( $# != 3 ))
 5 then
 6   echo "This script requires 3 arguments -- not $#."
 7   echo "Usage: $0 argument1 argument2 argument3"
 8   exit 1
 9 fi
10 echo "Correct number of parameters"
11 #
12 # The real code follows...
```

Minimum Number Of Arguments

```
 1 #!/bin/sh
 2 # validate02.sh
 3 #
 4 if (( $# < 1 ))
 5 then
 6   echo "This script requires 1 or more arguments."
 7   echo "Usage: $0 arg1 [arg2 ... argn]"
 8   exit 1
 9 fi
10 echo "$# arguments received."
11 #
12 # The real script follows ...
```

Numeric Arguments

```
#!/bin/bash
# Only a positive integer is required
function die () {
   echo >&2 "$@"
      exit 1
}
[ "$#" -eq 1 ] ||
   die "1 argument required, $# provided"
echo $1 | grep -E -q '^[0-9]+$' ||
   die "Numeric argument required, $1
   provided"
```

```
#!/bin/bash
# Be able to check all numeric formats
function isnum() {
    return `echo "$1" | awk -F"\n" '{print ($0 != $0+0)}'`
}
if isnum "$1"
then
  printf "%s is a number" $1
else
  printf "no it's not"
fi
```

File Type Validation

```
 1 #!/bin/sh
 2 # validate04.sh
 3 # Verify that a $1 is a valid input file.
 4 #
 5 if [[ ! -s "$1" ]]
 6 then
 7   echo "$1 does not exist or is empty"
 8   exit 1
 9 fi
10
11 if [[ ! -r "$1" ]]
12 then
13   echo "You do not have read permission for $1"
14   exit 2
15 fi
16 # The rest of the code follows...
```

Debugging Scripts

- Debug options included in the script

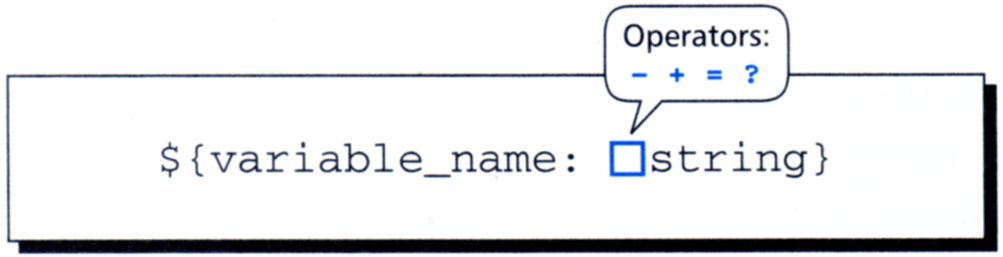
```
 1 #!/bin/sh
 2 # debug01.sh
 3 # Demonstrate debug options set in script
 4 #
 5 #set -o verbose
 6
 7 x=5
 8 (( y = x + 2 ))
 9
10 if (( y == 10 ))
11 then
12   echo \$y contains 10
13 else
14   echo \$y contains $y not 10
15 fi
16
17 while (( x != 0 ))
18 do
19   echo Counting down: \$x is $x
20   (( x = x - 1 ))
21 done
```

- Debug options on the command line
  - `bash -o xtrace  <scriptname>`

Variable Evaluation

- To access a variable
  - **`$var_name`**
  - **`${var_name}`**
    - cp $file $file.bak
    - cp $file ${file}.bak
    - Also be used in variable substitution
  - **`${!var_name}`**
    - Use the value of var_name as name of variable (indirect reference)

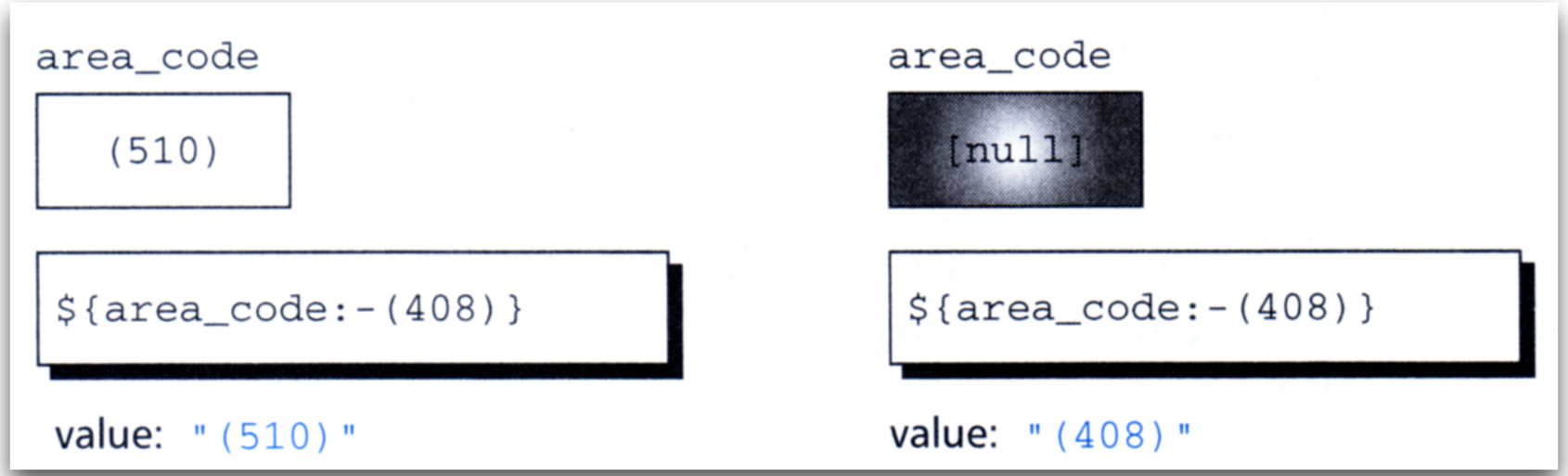
Variable Substitution



Operator	Description
${variable: –string}	If `variable` is not null, use the value of variable; otherwise, use `string`. The value of variable is not changed.
${variable: +string}	If `variable` is not null, use `string`; otherwise, use the value of the variable (null).
${variable: =string}	If `variable` is not null, use the value of variable; otherwise, use the value of `string` and assign the string to `variable`.
${variable: ?string}	If `variable` is not null, use the value of variable; otherwise, print the value of `string` and exit.

Substitute If Variable Null (-)

- `${var_name:-return_this_val_if_var_name_is_null}`



Substitute If Variable Not Null (+)

- `${var_name:+return_this_val_if_var_name_is_not_null}`

```
 1 #!/bin/bash
 2 # substitute-if-not-null.sh
 3 #
 4 dir=$(ls $1)
 5 echo ${dir:+"Directory $1 not empty. Can't delete it"}
 6
 7 if [[ -z "$dir" ]]
 8 then
 9   rmdir $1
10   echo $1 was removed
11 fi
```

Substitution Assignment (=)

- `${var_name:=initial_val_for_var_name_if_null}`

```
 1 #!/bin/bash
 2 # substitute-if-null.sh
 3 #
 4 echo "count contains: $count"
 5
 6 while (( ${count:=1} < 10 ))
 7 do
 8   echo "$count"
 9   (( count = count + 1 ))
10 done
```

Substitution Validation (?)

- `${var_name:?msg_to_stderr_if_var_name_is_null}`
  - Script exits after display the string to stderr

```
1 #!/bin/bash
2 # substitution-validation.sh
3 echo "Enter a value"
4 read val
5 test=${val:?"You did not enter a value: Script terminates"}
6
7 echo "Script continues"
```

Substring Expansion

- `${var_name:offset:length}`
  - Return parts of the string “var_name”, starting from character at location “offset” for “length” characters
    - “offset” starts from 0
  - Example
    - `count=frogfootman`
    - `${count:4} -> "footman"`
    - `${count:4:4} -> "foot"`

Pattern Matching in Bash

Pattern	Description
*	Match zero or more characters
?	Match any single character
[...]	Match any of the characters in a set
?(patterns)	Match zero or one occurrences of the patterns (extglob)
*(patterns)	Match zero or more occurrences of the patterns (extglob)
+(patterns)	Match one or more occurrences of the patterns (extglob)
@(patterns)	Match one occurrence of the patterns (extglob)
!(patterns)	Match anything that doesn't match one of the patterns (extglob)

Source: https://www.linuxjournal.com/content/pattern-matching-bash

More info: https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html

Pattern Matching I

- **`${var#pattern}`**
  - If the pattern matches the beginning of the variable's value, delete the shortest part that matches and return the rest
- **`${var##pattern}`**
  - Same as above but delete the longest match
- **`${var%pattern}`**
  - If the pattern matches the end of the variable's value, delete the shortest part that matches and return the rest
- **`${var%%pattern}`**
  - Same as above but delete the longest match

Pattern Matching: Example I

```
 1 #!/bin/bash
 2 # pattern-matching01.sh
 3 str="/usr/local/bin/myexefile.exe"
 4 echo str is \"$str\"
 5
 6 echo
 7 echo '${str#/*/}      = "'${str#/*/}'"'
 8 echo '${str##/*/}     = "'${str##/*/}'"'
 9 echo '${str}          = "'${str}'"'
10 echo '${str%.*}       = "'${str%.*}'"'
11 echo '${str%%.*}      = "'${str%%.*}'"'
12
13 echo
14 str="Come to class late is not good. I won't come to class late"
15 echo str is \"$str\"
16 echo '${str/class/the movie}  = "'${str/class/the movie}'"'
17 echo '${str//class/the movie} = "'${str//class/the movie}'"'
18
19 echo
20 echo '${str/#[Cclome/go}      = "'${str/#[Cclome/go}'"'
21 echo '${str/%late/early}      = "'${str/%late/early}'"'
```

Pattern Matching II

- **`${var/pattern/string}`**
  - The first longest match to *pattern* in *var* is replaced by *string*
- **`${var/[/#%]pattern/string}`**
  - If *pattern* begins with a /, all longest matches to pattern in var are replaced by string
  - If *pattern* begins with a #, it must match at the start (begin) of *var*
  - If *pattern* begins with a %, it must match at the end of *var*
  - If *string* is *null*, the matches are deleted
  - If *var* is @ or *, the operation is applied to each positional parameter in turn and the expansion is the resultant list

Pattern Matching: Example II

```
 1 #!/bin/bash
 2 # pattern-matching02.sh
 3 if (( $# <= 2 ))
 4 then
 5   echo "Usage: $0 file1 file2 file3 ..."
 6   echo "  files must end with .jpg"
 7   exit 1
 8 fi
 9
10 echo 'You entered parameters: "'$*'"'
11 echo "I\'ll change to: \""${*/%.jpg/.png}'"'
```

Miscellaneous Operators

- Length operator
  - `${#var}`
    - Returns the length of the value of var as a string
- Command substitution
  - `$(command)`
    - Output from stdout of the command is returned
    - Backward compatible with backquotes: `` `cmd` ``

Arrays

- Array indexes
  - Starts from 0 (zero)
  - Doesn't have to be consecutive - in some sense, similar to "associative array"
  - E.g., `names[0], name[2]`

- Initialization
  - `names[2]=alice; name[0]=hatter; name[1]=bo`
  - `names=([2]=alice [0]=hatter [1]=bo)`
  - `names=(hatter bo alice)`
  - `names=(hatter [5]=bo alice)`

Arrays (ii)

- Accessing/referencing
  - `${names[index]}`
  - `@` and `*` can be used and work in the same way as for positional parameters
  - `${#names[@]}` returns # of elements

```
 1 #!/bin/bash
 2 # array01.sh
 3 names=(aa bb cc dd ee)
 4 for i in "${names[@]}"; do
 5   echo $i
 6 done
 7 echo
 8 for i in "${names[*]}"; do
 9   echo $i
10 done
11
12 echo Number of elements of names[] is ${#names[@]}
13 echo Number of characters in names[0] is ${#names}
```

Arrays (iii)

- A list of defined indexes
  - `${!array[@]}`
- Delete an element
  - `unset array[n]`
- Delete the entire array
  - `unset array` or
    `unset array[@]` or
    `unset array[*]`

```
 1 #!/bin/bash
 2 # array03.sh
 3
 4 for i in $(cut -f 1,3 -d: /etc/passwd) ; do
 5   echo i is $i
 6   echo '[${i#*:}] is ' ${i#*:}
 7   echo '[${i%:*}] is ' ${i%:*}
 8   array[${i#*:}]=${i%:*}
 9 done
10
11 echo "User ID $1 is ${array[$1]}."
12 echo "There are currently ${#array[*]} user accounts on the system."
```

Shell Script I/O

- File-oriented I/O
  - Redirections
  - Pipes
  - echo/printf
- Interactive I/O
  - read
  - echo/printf
- File-oriented I/O is more preferable
  - Due to its automaticity in nature

echo

Table 7-3. echo escape sequences

Sequence	Character printed
\a	ALERT or CTRL-G (bell)
\b	BACKSPACE or CTRL-H
\c	Omit final NEWLINE
\e	Escape character (same as \E)
\E	Escape character[a]
\f	FORMFEED or CTRL-L
\n	NEWLINE (not at end of command) or CTRL-J
\r	RETURN (ENTER) or CTRL-M
\t	TAB or CTRL-I
\v	VERTICAL TAB or CTRL-K
n	ASCII character with octal (base-8) value *n*, where *n* is 1 to 3 digits
\0*nnn*	The eight-bit character whose value is the octal (base-8) value *nnn* where *nnn* is 1 to 3 digits
\x*HH*	The eight-bit character whose value is the hexadecimal (base-16) value *HH* (one or two digits)
\\	Single backslash

[a] Not available in versions of *bash* prior to 2.0.

echo

Table 7-2. echo options

Option	Function
-e	Turns on the interpretation of backslash-escaped characters
-E	Turns off the interpretation of backslash-escaped characters on systems where this mode is the default
-n	Omits the final newline (same as the \c escape sequence)

- Examples
  - `echo -n A string to be displayed`
  - `echo Here is a \t of tab`
  - `echo Here is a \\t of tab`
  - `echo -e Here is a \\t of tab`
  - Note: the shell steals a *backslash* before passing arguments to **echo**

printf

- C-style printf
- Additional bash printf specifiers
  - `%b`
    - Expands echo-style escape sequences in the argument string
      - `printf "%s\n" 'hello\nworld'`
  - `%q`
    - Prints the string argument in such a way that it can be used for shell input
      - `printf "%q\n" 'greetings to the world'`

read

- **`read var1 var2 ...`**
  - Takes a line from stdin and breaks it down into words delimited by "IFS", words are assigned to var1, var2, ...
  - If vars are omited, the entire line is assigned to `$REPLY`
  - read's exit status is 1, when nothing to read (normally return 0)
  - Options
    - **`-a`**: array
      - Example: **`read -a people`**

** read is inefficient compared to redirections, only use it with small input files **

Process ID and Job Numbers

- PID
  - Assigned by OS
  - Refer to all processes running in the system
  - ps –f, ps –ef
- Job ID
  - Assigned by your shell
  - Refer to background processes
  - jobs –l
    - also list processID
  - jobs –p
    - only list processID
  - jobs –x <command>
    - execute <command>
    - %n can be used to refer to jobID 'n'

Signals

- Signal
  - A message that one process sends to another
    - when an abnormal event occurs or
    - when it wants the other process to do something
  - Shell built-in command to send a signal
    - kill
  - We can refer to a signal using
    - Numbers
    - Names
    - Check out `man 7 signal` for signal list
  - Practice
    - Write a script called `killalljobs` that kills all background jobs
    -

Process ID variables

- $$
  - process ID of the current shell
- $!
  - process ID of the most recently invoked background job
- Practice
  - Write a script to test these two variables

trap

- Let shell script reacts to a specific signal
- Syntax
  - trap cmd sig1 sig2 ...
- Example

```
#!/bin/bash
trap "echo 'Receive SIGINT'" SIGINT
trap "echo 'Receive SIGUSR1'" SIGUSR1

while true; do
   sleep 60
done
```

- Usage
  - Clean up temporary files

trap: Clean up temporary files

```
#!/bin/bash

cleanup(){
  if [ -e $msgfile ]; then
    rm $msgfile
  fi
  exit
}

trap cleanup SIGINT SIGTERM

msgfile=/tmp/msg$$
cat > $msgfile
sleep 15
rm $msgfile
```

wait

- wait <child-pid>
  - make the script waits for "background child process" to finish
  - child process's exit status can be collected via $?

```
 1 #!/bin/bash
 2 # main.sh
 3 #
 4 echo "Run script 1" && ./1.sh &
 5 process_id=$!
 6 echo "Run script 2" && ./2.sh &
 7
 8 wait $!
 9 echo script 2.sh is done with exit status $?
10
11 wait $process_id
12 echo script 1.sh is done with exit status $?
```

```
lrwxrwxrwx 1 user user 4 Oct  6 04:31 2.sh -> 1.sh
```

```
 1 #!/bin/bash
 2 # 1.sh
 3 #
 4 let exit_code=$RANDOM%254+1
 5 echo $0-$$ Prepared exitcode = $exit_code
 6 echo $0-$$ is started
 7 echo $0-$$ PID is $$
 8 echo -n $0-$$ sleep 1 sec for 5 times
 9 for i in 1 2 3 4 5
10 do
11   echo $0-$$#$i" "
12   sleep 1
13 done
14 echo $0-$$ is exiting and return $exit_code
15 exit $exit_code
```

Subshells

- Subshells
  - whenever we run a shell script, we actually invoke another copy of the shell
  - Practice
    - Write a simple shell script to test this fact
  - What are inherited from parent shell
    - current directory
    - environment variables
    - stdin, stdout, stderr + all open file descriptors
    - signals that are ignored
  - What are not…
    - Shell variables
    - handling of signals that are not ignored