Bash programming

Dr. B. Bharathi

SSNCE

September 07, 2019

BASIC SHELL PROGRAMMING

- A script is a file that contains shell commands
  - data structure: variables
  - control structure: sequence, decision, loop
- Shebang line for bash shell script:

  `#! /bin/bash`

  `#! /bin/sh`
- to run:
  - make executable: **`% chmod +x script`**
  - invoke via: **`% ./script`**

BASH SHELL PROGRAMMING

- Input
  - prompting user
  - command line arguments
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions
- Traps

USER INPUT

- shell allows to prompt for user input

Syntax:

```
read varname [more vars]
```

- or

```
read –p "prompt" varname [more vars]
```

- words entered by user are assigned to **`varname`** and "**`more vars`**"
- last variable gets rest of input line

USER INPUT EXAMPLE

```
#! /bin/sh
read -p "enter your name: " first last

echo "First name: $first"
echo "Last name: $last"
```

SPECIAL SHELL VARIABLES

Parameter	Meaning
$0	Name of the current shell script
$1-$9	Positional parameters 1 through 9
$#	The number of positional parameters
$*	All positional parameters, “$*” is one string
$@	All positional parameters, “$@” is a set of strings
$?	Return status of most recently executed command
$$	Process id of current process

EXAMPLES: COMMAND LINE ARGUMENTS

```
% set tim bill ann fred
      $1  $2   $3  $4
% echo $*
tim bill ann fred
% echo $#
4
% echo $1
tim
% echo $3 $4
ann fred
```

The ‘set’ command can be used to assign values to positional parameters

BASH CONTROL STRUCTURES

- if-then-else
- case
- loops
  - for
  - while
  - until
  - select

IF STATEMENT

```
if command
then
  statements
fi
```

- statements are executed only if **`command`** succeeds, i.e. has return status “0”

TEST COMMAND

Syntax:

```
test expression
[ expression ]
```

- evaluates ‘expression’ and returns true or false

Example:

```
if test –w "$1"
  then
  echo "file $1 is write-able"
fi
```

THE SIMPLE IF STATEMENT

```
if [ condition ]; then
  statements
fi
```

- executes the statements only if **`condition`** is true

THE IF-THEN-ELSE STATEMENT

```
if [ condition ]; then
      statements-1
else
  statements-2
fi
```

- executes statements-1 if condition is true
- executes statements-2 if condition is false

THE IF…STATEMENT

```
if [ condition ]; then
    statements
elif [ condition ]; then
    statement
else
    statements
fi
```

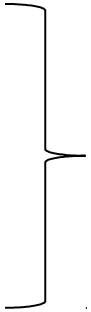
- The word **elif** stands for “else if”
- It is part of the if statement and cannot be used by itself

RELATIONAL OPERATORS

Meaning	Numeric	String
Greater than	-gt	
Greater than or equal	-ge	
Less than	-lt	
Less than or equal	-le	
Equal	-eg	= or ==
Not equal	-ne	!=
str1 is less than str2		str1 < str2
str1 is greater str2		str1 > str2
String length is greater than zero		-n str
String length is zero		-z str

COMPOUND LOGICAL EXPRESSIONS

!	not
&&	and
||	or



and, or
must be enclosed within

[[ ]]

EXAMPLE: USING THE ! OPERATOR

```
#!/bin/bash

read -p "Enter years of work: " Years
if [ ! "$Years" -lt 20 ]; then
   echo "You can retire now."
else
   echo "You need 20+ years to retire"
fi
```

EXAMPLE: USING THE && OPERATOR

```
#!/bin/bash

Bonus=500
read -p "Enter Status: " Status
read -p "Enter Shift: " Shift
if [[ "$Status" = "H" && "$Shift" = 3 ]]
then
   echo "shift $Shift gets \$$Bonus bonus"
else
   echo "only hourly workers in"
   echo "shift 3 get a bonus"
fi
```

EXAMPLE: USING THE || OPERATOR

```
#!/bin/bash

read -p "Enter calls handled:" CHandle
read -p "Enter calls closed: " CClose
if [[ "$CHandle" -gt 150 || "$CClose" -gt 50 ]]
   then
   echo "You are entitled to a bonus"
else
   echo "You get a bonus if the calls"
   echo "handled exceeds 150 or"
   echo "calls closed exceeds 50"
fi
```

FILE TESTING

	Meaning
-d file	True if ‘file’ is a directory
-f file	True if ‘file’ is an ord. file
-r file	True if ‘file’ is readable
-w file	True if ‘file’ is writable
-x file	True if ‘file’ is executable
-s file	True if length of ‘file’ is nonzero

EXAMPLE: FILE TESTING

```
#!/bin/bash
echo "Enter a filename: "
read filename
if [ ! –r "$filename" ]
 then
   echo "File is not read-able"
 exit 1
fi
```

EXAMPLE: FILE TESTING

```
#! /bin/bash

if [ $# -lt 1 ]; then
        echo "Usage: filetest filename"
        exit 1
fi
if [[ ! -f "$1" || ! -r "$1" || ! -w "$1" ]]
then
  echo "File $1 is not accessible"
  exit 1
fi
```

EXAMPLE: IF… STATEMENT

```
# The following THREE if-conditions produce the same result

* DOUBLE SQUARE BRACKETS
read -p "Do you want to continue?" reply
if [[ $reply = "y" ]]; then
   echo "You entered " $reply
fi

* SINGLE SQUARE BRACKETS
read -p "Do you want to continue?" reply
if [ $reply = "y" ]; then
   echo "You entered " $reply
fi

* "TEST" COMMAND
read -p "Do you want to continue?" reply
if test $reply = "y"; then
   echo "You entered " $reply
fi
```

EXAMPLE: IF..ELIF... STATEMENT

```
#!/bin/bash

read -p "Enter Income Amount: " Income
read -p "Enter Expenses Amount: " Expense

let Net=$Income-$Expense

if [ "$Net" -eq "0" ]; then
   echo "Income and Expenses are equal -
  breakeven."
elif [ "$Net" -gt "0" ]; then
   echo "Profit of: " $Net
else
   echo "Loss of: " $Net
fi
```


THE CASE STATEMENT

- use the case statement for a decision that is based on multiple choices

Syntax:

```
case word in
    pattern1) command-list1
    ;;
    pattern2) command-list2
    ;;
    patternN) command-listN
    ;;
esac
```

CASE PATTERN

- checked against word for match
- may also contain:

  `*`

  `?`

  `[ … ]`

  `[:class:]`
- multiple patterns can be listed via:

  `|`

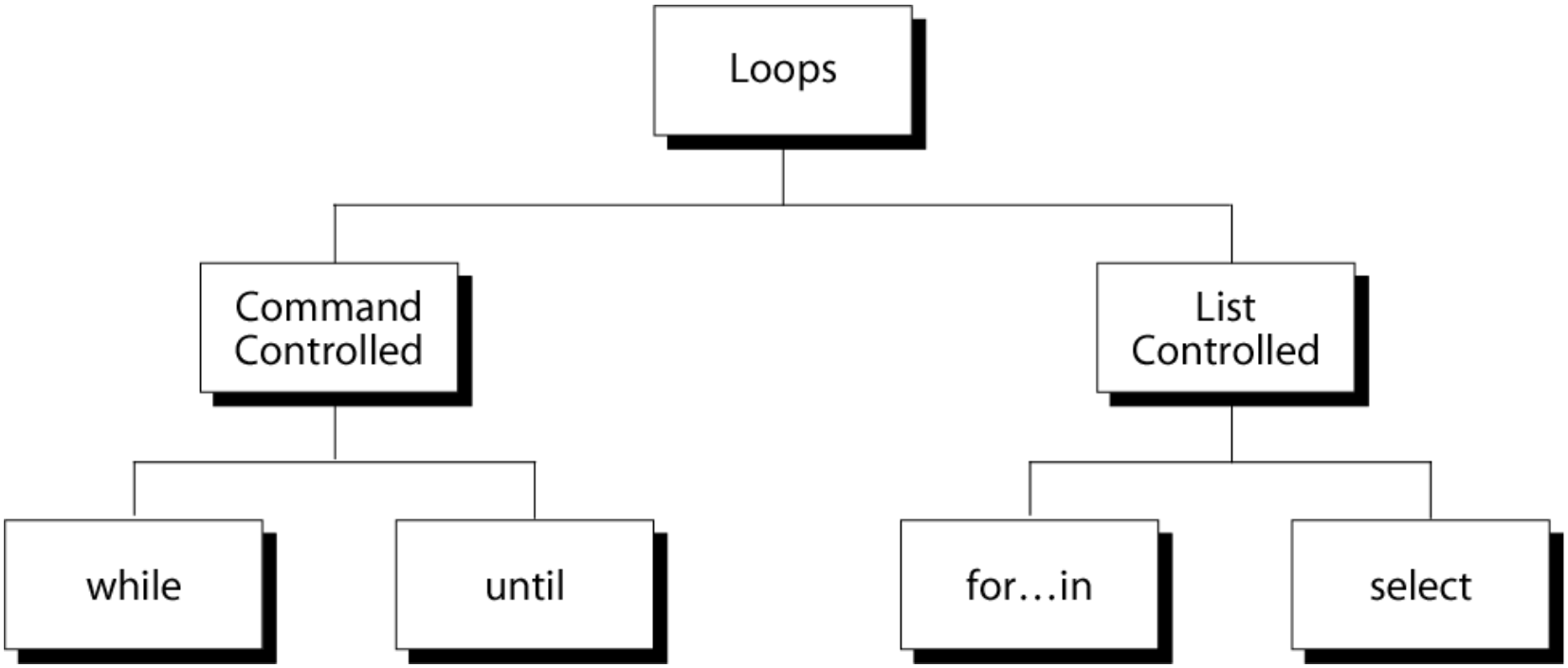
EXAMPLE 1: THE CASE STATEMENT

```
#!/bin/bash
echo "Enter Y to see all files including hidden files"
echo "Enter N to see all non-hidden files"
echo "Enter q to quit"

read -p "Enter your choice: " reply

case $reply in
  Y|YES) echo "Displaying all (really…) files"
         ls -a ;;
  N|NO)  echo "Display all non-hidden files..."
         ls ;;
  Q)     exit 0 ;;

  *) echo "Invalid choice!"; exit 1 ;;
esac
```

EXAMPLE 2: THE CASE STATEMENT

```
#!/bin/bash
ChildRate=3
AdultRate=10
SeniorRate=7
read -p "Enter your age: " age
case $age in
  [1-9]|[1][0-2])   # child, if age 12 and younger
     echo "your rate is" '$'"$ChildRate.00" ;;
  # adult, if age is between 13 and 59 inclusive
  [1][3-9]|[2-5][0-9])
     echo "your rate is" '$'"$AdultRate.00" ;;
  [6-9][0-9])       # senior, if age is 60+
     echo "your rate is" '$'"$SeniorRate.00" ;;
esac
```

REPETITION CONSTRUCTS



THE WHILE LOOP

- Purpose:

  To execute commands in “command-list” as long as “expression” evaluates to true

Syntax:

```
while [ expression ]
do
    command-list
done
```

EXAMPLE: USING THE WHILE LOOP

```
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 10 ]
do
   echo The counter is $COUNTER
   let COUNTER=$COUNTER+1
done
```

EXAMPLE: USING THE WHILE LOOP

```
#!/bin/bash

Cont="Y"
while [ $Cont = "Y" ]; do
  ps -A
  read -p "want to continue? (Y/N)" reply
  Cont=`echo $reply | tr [:lower:] [:upper:]`
done
echo "done"
```

EXAMPLE: USING THE WHILE LOOP

```
#!/bin/bash
# copies files from home- into the webserver- directory
# A new directory is created every hour

PICSDIR=/home/carol/pics
WEBDIR=/var/www/carol/webcam
while true; do
   DATE=`date +%Y%m%d`
   HOUR=`date +%H`
   mkdir $WEBDIR/"$DATE"
   while [ $HOUR -ne "00" ]; do
      DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
      mkdir "$DESTDIR"
      mv $PICSDIR/*.jpg "$DESTDIR"/
      sleep 3600
      HOUR=`date +%H`
   done
done
```

THE UNTIL LOOP

- Purpose:

  To execute commands in “command-list” as long as “expression” evaluates to false

Syntax:

```
until [ expression ]
do
    command-list
done
```

EXAMPLE: USING THE UNTIL LOOP

```
#!/bin/bash

COUNTER=20
until [ $COUNTER -lt 10 ]
do
   echo $COUNTER
   let COUNTER-=1
done
```

EXAMPLE: USING THE UNTIL LOOP

```
#!/bin/bash

Stop="N"
until [ $Stop = "Y" ]; do
  ps -A
  read -p "want to stop? (Y/N)" reply
  Stop=`echo $reply | tr [:lower:] [:upper:]`
done
echo "done"
```

THE FOR LOOP

- Purpose:

  To execute commands as many times as the number of words in the “argument-list”

Syntax:

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

EXAMPLE 1: THE FOR LOOP

```
#!/bin/bash

for i in 7 9 2 3 4 5
do
   echo $i
done
```

EXAMPLE 2: USING THE FOR LOOP

```
#!/bin/bash
# compute the average weekly temperature

for num in 1 2 3 4 5 6 7
do
   read -p "Enter temp for day $num: " Temp
   let TempTotal=$TempTotal+$Temp
done

let AvgTemp=$TempTotal/7
echo "Average temperature: " $AvgTemp
```


38

LOOPING OVER ARGUMENTS

- simplest form will iterate over all command line arguments:

```
#! /bin/bash
for parm
do
        echo $parm
done
```

SELECT COMMAND

- Constructs simple menu from word list
- Allows user to enter a number instead of a word
- User enters sequence number corresponding to the word

Syntax:

```
select WORD in LIST
do
    RESPECTIVE-COMMANDS
done
```

- Loops until end of input, i.e. ^d (or ^c)

SELECT EXAMPLE

```
#! /bin/bash
select var in alpha beta gamma
do
      echo $var
done
```

- Prints:

```
1) alpha
2) beta
3) gamma
#? 2
beta
#? 4
#? 1
alpha
```

SELECT DETAIL

- PS3 is select sub-prompt
- $REPLY is user input (the number)

```
#! /bin/bash
PS3="select entry or ^D: "
select var in alpha beta
do
     echo "$REPLY = $var"
done
```

Output:

```
select ...
1) alpha
2) beta
? 2
2 = beta
? 1
1 = alpha
```

SELECT EXAMPLE

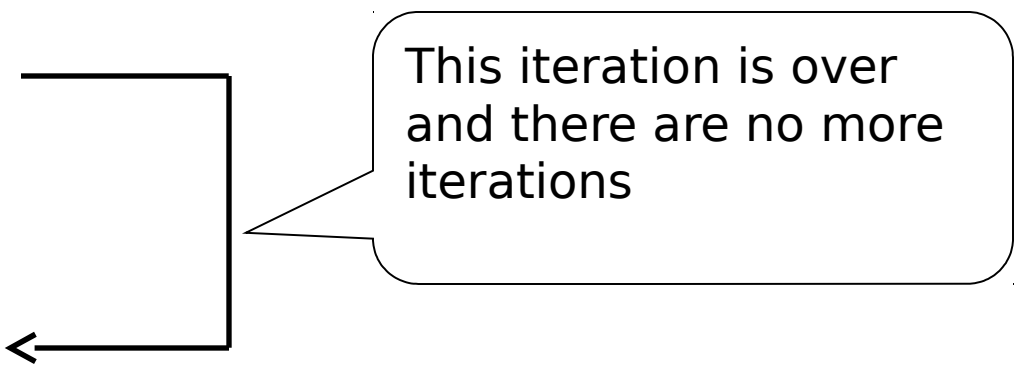
```
#!/bin/bash
echo "script to make files private"
echo "Select file to protect:"

select FILENAME in *
do
  echo "You picked $FILENAME ($REPLY)"
  chmod go-rwx "$FILENAME"
  echo "it is now private"
done
```

BREAK AND CONTINUE

- Interrupt for, while or until loop
- The break statement
  - transfer control to the statement AFTER the done statement
  - terminate execution of the loop
- The continue statement
  - transfer control to the statement TO the done statement
  - skip the test statements for the current iteration
  - continues execution of the loop

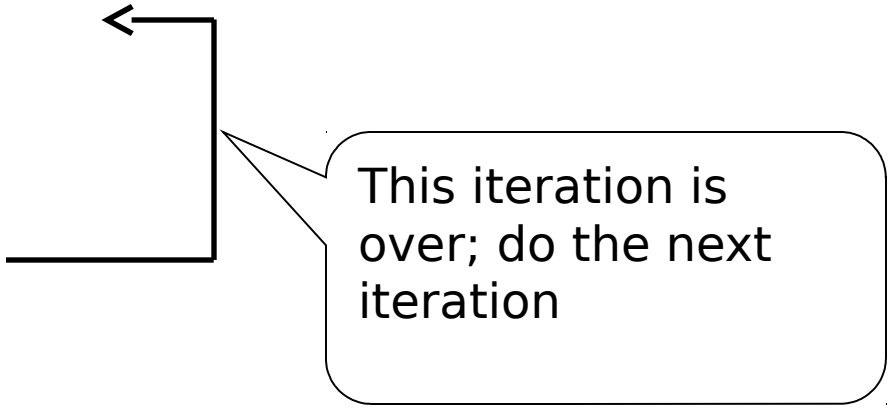
THE BREAK COMMAND

```
while [ condition ]
do
    cmd-1
    break
    cmd-n
done
echo "done"
```



THE CONTINUE COMMAND

```
while [ condition ]
do
    cmd-1
    continue
    cmd-n
done
echo "done"
```



EXAMPLE:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
        if [ $index –le 3 ]; then
             echo "continue"
             continue
        fi
        echo $index
        if  [ $index –ge 8 ]; then
             echo "break"
             break
        fi
done
```

BASH SHELL PROGRAMMING

- Sequence
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions

DONE !

- Traps

still to come

SHELL FUNCTIONS

- A shell function is similar to a shell script
  - stores a series of commands for execution later
  - shell stores functions in memory
  - shell executes a shell function in the same shell that called it
- Where to define
  - In .profile
  - In your script
  - Or on the command line
- Remove a function
  - Use unset built-in

SHELL FUNCTIONS

- must be defined before they can be referenced
- usually placed at the beginning of the script

Syntax:

```
function-name () {
   statements
}
```

EXAMPLE: FUNCTION

```
#!/bin/bash

funky () {
  # This is a simple function
  echo "This is a funky function."
  echo "Now exiting funky function."
}

# declaration must precede call:

funky
```

EXAMPLE: FUNCTION

```
#!/bin/bash
fun () { # A somewhat more complex function.
  JUST_A_SECOND=1
  let i=0
  REPEATS=30
  echo "And now the fun really begins."
  while [ $i -lt $REPEATS ]
  do
     echo "-------FUNCTIONS are fun-------->"
     sleep $JUST_A_SECOND
     let i+=1
  done
}
fun
```

FUNCTION PARAMETERS

- Need not be declared
- Arguments provided via function call are accessible inside function as $1, $2, $3, …

$# reflects number of parameters

$0 still contains name of script
(not name of function)

EXAMPLE: FUNCTION WITH PARAMETER

```
#! /bin/sh
testfile() {
  if [ $# -gt 0 ]; then
     if [[ -f $1 && -r $1 ]]; then
        echo $1 is a readable file
     else
        echo $1 is not a readable file
     fi
  fi
}

testfile .
testfile funtest
```

EXAMPLE: FUNCTION WITH PARAMETERS

```
#! /bin/bash
checkfile() {
   for file
   do
      if [ -f "$file" ]; then
         echo "$file is a file"
      else
         if [ -d "$file" ]; then
            echo "$file is a directory"
         fi
      fi
   done
}
checkfile . funtest
```

LOCAL VARIABLES IN FUNCTIONS

- Variables defined within functions are global,
  i.e. their values are known throughout the entire shell program

- keyword “local” inside a function definition makes referenced variables “local” to that function

EXAMPLE: FUNCTION

```
#! /bin/bash

global="pretty good variable"

foo () {
        local inside="not so good variable"
        echo $global
        echo $inside
        global="better variable"
}

echo $global
foo
echo $global
echo $inside
```