1. Bash Shell Features (Update 1)

shell builtins, redirection, operators, variables, functions

bash Features

- Command Interpreter, Processor and Language (for rapid prototyping)
- Customized environments via (.bash_profile, .bashrc) initialization files
- Capture frequently used commands via history, aliases, scripts and functions
- Uses scripts for replicating commands repeatedly on multiple files
- Common user environment by System Administrators
- Allows periodic, scheduled tasks in scripts to run
- Does Command Completion

bash Features (2)

- Unique bash facilities:
  - long (word) options [ls --help]
  - POSIX mode & conformance [e.g. printf ; set -o posix {or --posix}]
  - Regex Character Classes { [[:alpha:]] }
  - Command arithmetic
  { for ((expr1;expr2; expr3)); do commands; done }
  - functions, variables share name space
  - $'…' and $"…" quoting for strings
  - Arrays of unlimited size
  - '!' reserved word
  - '**' arithmetic exponentiation operator
  - Redirection '&>' for STDOUT and STDERR (= > file 2>&1)
  - Prompt ($PS1) expansion with backslash escapes and command substitution
  - here string input redirection '<<<' facility

- See <tiswww.case.edu/php/chet/bash/FAQ>

bash Responsibilities

- Run Startup files, set global variable values
- Interpret the commandline
- Do variable substitution
- do file name expansion (wild cards)
- Set up I/O redirection
- Set up unnamed pipes between commands
- Execute commands and programs
- Execute complete, built-in interpreted programming language scripts

Sample Command Manipulations

- Delay scripts with **sleep, wait**
  - $ **sleep {*No. of seconds*}; command/script**
  - $ **wait [*process id*]; command/script**
- Schedule scripts with **at** (**cron** shown elsewhere)
  - $ **at [-t timeformat] -f ./myscript**
- Repeat scripts with **watch** and **!#** Event Designator
  - $ **watch -n 5 free -m # 5 second repeats**
  - $ **watch -d 'ls -l | grep -F katz' \**
    # show differences each time
  - $ **command/script; #!   # repeats current line once**

2. Keyboard MetaCharacters

^C ^D ^Z <ESC> ^V ^H ^?

Keyboard Shortcuts (vim)

- Enable vim keyboard shortcuts:
  $ set -o vi # all vim commands \
  work on current commandline

- ~/.bash_history history list allows command reuse.

- <Ctrl-Z> Suspend foreground command; fg resumes it.
  <Ctrl-C | \> Kill current job (not bash)
  <Ctrl-D> Kill current login session
  <Ctrl-H> Erase last Character
  <Ctrl-W> Erase last Word
  <Ctrl-?> Erase line so far
  <Ctrl-S> Stop (Freeze) output
  <Ctrl-Q> Start (unfreeze) output
  <Ctrl-V> Take next char literally

command	meaning
[<esc>]{- or j}	go up history list
{+ or k}	go down history list
h, l	move cursor left, right
A	Insert at end of line
0, $	go to 1st, last character
i, a	insert before, after cursor
x	Delete Char under cursor
cw	Change Word
<Ctrl-T>	swap last 2 chars.
u	undo last shortcut

stty and tset Commands

- **stty (1)** Displays or changes terminal line settings.
  $ **stty -a # shows all settings**
- **tty (1)** Shows what tty port you are connected to
- $ **stty sane # resets terminal to default settings**
- **tset (1)** initializes terminals based on terminal type.
- **To restore terminal functionality, type:**
  <Ctrl-J>tset|reset<Ctrl-J> # <Ctrl-J> is a line feed
- **To reset the environment variable TERM, type:**
  $ **eval `tset -s`   # Can also put in .bash_profile**

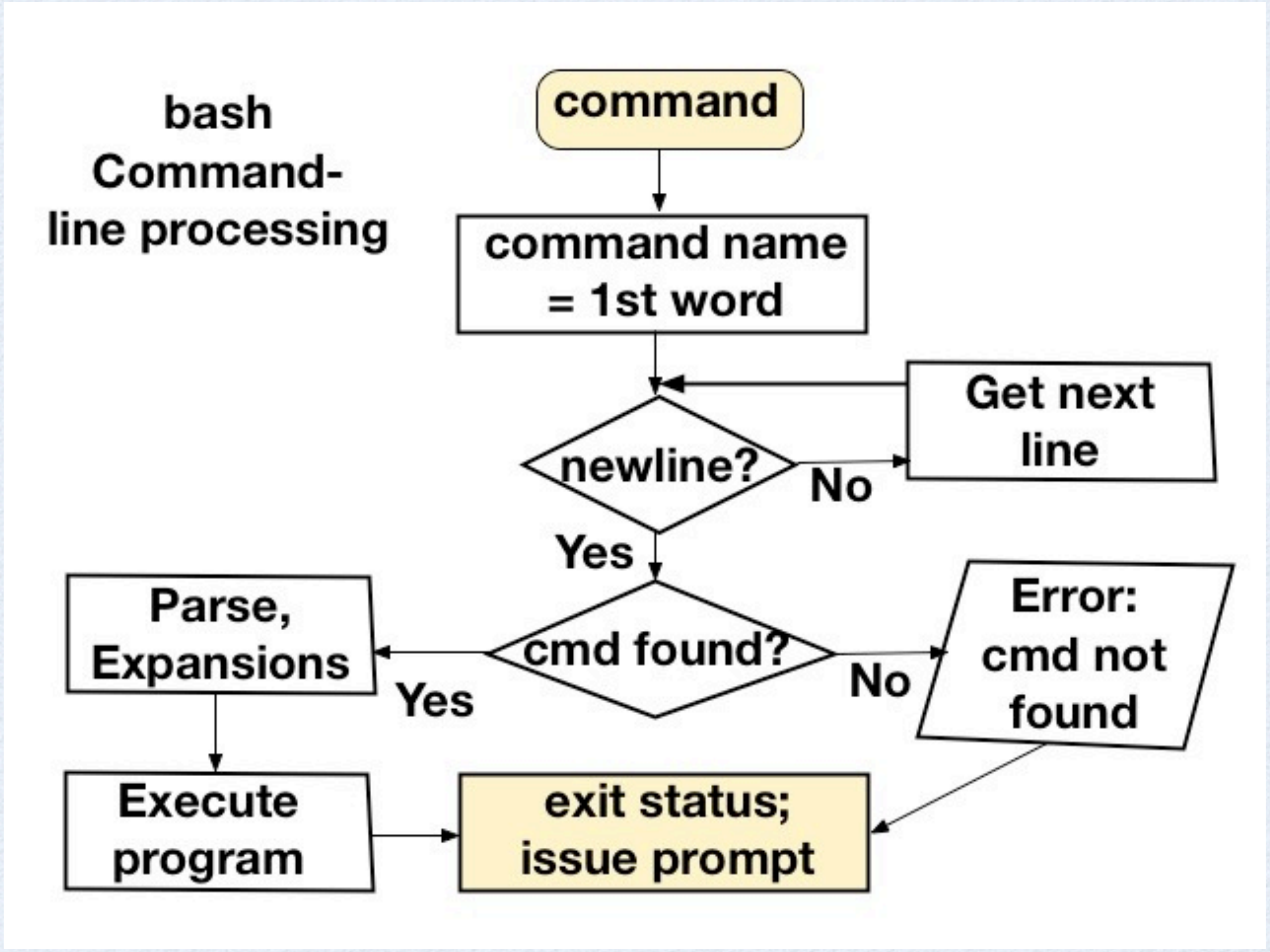
8

Commands as Symbols

Symbol	Synonym	Command Meaning
( )	bash	Start a subshell within a commandline as a group of commands
$( )	`command`	Command Substitution
(( ))	let	Arithmetic evaluation; expression includes an '=' sign
$(( ))	`  `	Arithmetic expansion (excludes '=' sign) with substitution of result
[ ]	test	Test arithmetic or relational expression as true or not
[[ ]]	test	Test arithmetic, string or relational expression as true or not

3. Customizing bash

bash Commandline processing



Initialization (Startup) Files

- There are 3 kinds of bash shells:
  o **Interactive login** shell [note: -bash in ps] via (virtual) console or via ssh
  o **Interactive non-login shell** via gnome or kde terminal
  o **Non-interactive shell** or **subshell** [scripts, invoking a subshell]

- The login process looks for startup files for all users containing commands in **/etc/profile, /etc/inputrc, /etc/profile.d/*bash*** and customized for you in **~/.bash_profile, ~/.bash_login, or ~/.profile.** For a subshell, **~/.bashrc** is run

- When you logout, bash issues commands in **~/.bash_logout** [e.g. cleanup and temp file removal]

bash Aliases U1

- alias is a (short) command name for a commandline
- Form: $ alias [name[='commandline']]
  Alt Form: $ alias [name[="commandline"]]
  # Use this for variable and command substitution
  Antidote: $ unalias name
- An Alias never replaces itself, but: $ alias ls='ls -Fa'
- Aliases can be nested: $ alias lssum="ll | wc -l"
- To temporarily suspend an alias, (e.g. ls) use:
  $ \ls or $ /fullpath/ls
- Example: $ alias r='fc -s '
  $ alias lss='ls -las '
  $ r lss # repeats last command starting with lss

13

4. bash Variables

var $var ${var} ${array[*]} PATH PS1 SHELL TERM

14

Environment Variables

- **Variable** = a named container of (string) data (single value). **Environment** (**global**) (uppercase) Variables with values available in (login) shell on down; **Local** (lowercase) variables with values available only in shell they are defined in.

- **Variable Names**: 1st character **[A-Za-z_]**; other characters **[A-Za-z0-9_]**
  Define by name; Reference with $ prefix. (**var=1; echo $var**)
  Note: setting a variable only for a script: $ **var=1 script.bash**

- Defined variables are local unless exported.
  $ **var="one two three" ; read newvar # [local]**
  $ **echo $var ${newvar}   # display variable value**
  $ **export var newvar # global in future subshells**
  $ **export var="four five six" >> ~/.bash_profile #global, in all future Login shells (and on down)**

Environment Variables (2)

- Nullify value of variable
  $ unset $newvar   # remove variable value, set it to null but retain variable name

- Make variable definitions available for all login sessions
  $. ~/.bash_profile # same as: source ~/.bash_profile
  # Run the above command instead of logging out and back in

- $ env | less # view current values of global variables
  $ declare -p | less # view names of all variables and their scope

Customizing Primary Prompt

- Primary Prompt initial setting in /etc/bash.bashrc

```
PS1="${USER}@${HOST}:${PWD}> "
$ echo $PS1
katz@linux-lwsr:~>
```

- Customizing PS1 in ~/.bash_profile

```
export PS1="\[$(ppwd)\]\u@\h:\w [\!] >"
BLUE="\[\e[1;34m\]"; NORMAL="\[\e[0m\]";
RED="\[\e[1;31m\]"
export PS1="\[$(ppwd)\]$BLUE\u$NORMAL@\h:
$RED\w$NORMAL [\!] >"
$ echo $PS1
katz@linux-lwsr:~ [331]>
```

Global Variable Meanings

- PATH=/home/katz/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11
  # only directories bash will look for command names
  PATH=$PATH:new_dir # appends new_dir to PATH
- HOSTNAME=linux-lwsr.site
- SHELL=/bin/bash
- TERM=xterm
- LOGNAME=katz
- PWD=/home/katz
- _=/usr/bin/env   # last word of last command
- PS1="\u@\h \W [\!] \$ "   # See previous slide

Quoting

- \ makes the next character ordinary
  - \$ makes $ not special; (\\ becomes \ )
- '...' prevents any bash interpretation
- "..." prevents any bash interpretation except variable evaluation, command substitution and backslashes (\)
- \`...\` { back quotes } or $(...) command substitution executes the command within and its result replaces the back quotes. bash then runs the entire modified commandline. Use \\\` ... \\\` for 1 level of nesting.

Special Variables

- Variables can have 3 states:
  - it doesn't exist, [=disabled or unset]
  - it exists, but is empty ("") [=enabled or set]
  - it exists, and is not empty [=enabled or set]

- **Positional Parameters (Commandline arguments):**
  $0 [= shell/script name] $1 $2 ... $9 ${10} ${11}...
  Assign values via builtin **set** or in script arguments
  $ **set -- hi there how are you? ; echo $0 $1 $2 $3 $4 $5**

- **Special Parameters: $#** argument list count
  $*** concatenated arguments **$@ same as **$*** but quoted args
  $!** last background Process ID **$$ current Process ID
  $_ rightmost word (non-command) of previous line
  $- shows options of the session login shell

Special Variables (2)

- **String Operators in Variables**
  ${var:-word} var exists, not null, value, else word
  ${var:+word} var exists, not null, word else null
  ${var:=word} var exists, not null, value, else var=word (persists)
  ${var:?[mesg]} var exists, not null, word else error message
  ${var:offset:length} return substring starting at offset and up to length characters
  ${#var} the number of characters in var's value is output

- Examples: $ **echo $var; echo ${var:-A1} # outputs A1**
  $ **var=25 echo $var; echo ${var:+true} # outputs 25 true**
  $ **var="" echo $var; echo ${var:?"not set"} # outputs not set**
  $ **var=abcdefg echo $var; echo ${var:2:4} # outputs abcdefg**

Special Variables (3)

- Pattern Matching String Operators **(? * [ ] wildcards used)**
  ${var#pattern} output var value minus shortest beginning pattern
  ${var##pattern} output var value minus longest beginning pattern
  ${var%pattern} output var value minus shortest ending pattern
  ${var%%pattern} output var value minus longest ending pattern
  ${var/pattern/string} longest match to pattern in value is is replaced by string once (#, % used as anchors)
  ${var//pattern/string} longest match to pattern in value is is replaced by string for all matches (#, % used as anchors)

Special Variables (4)

U1

- Examples:

```
$ var=/home/katz/long.file.name
$ echo ${var#/h*/}  # outputs katz/long.file.name
$ echo ${var##/h*/}  # outputs long.file.name
$ echo ${var%.*e}  # outputs /home/katz/long.file
$ echo ${var%%.*e}   # outputs /home/katz/long
$ echo ${var/[aeiou]/X}
/hXme/katz/long.file.name
$ echo ${var//[aeiou]/X}
/hXmX/kXtz/lXng.fXlX.nXmX
```

23

bash Type Variables

- **declare** builtin command options

-a	variable is an indexed array
-A	variable is an associative array
-f	name is a function, not a variable
-i	variable is an integer
-r	variable is a constant (readonly)
-x	variable is global (exported)

- List each variable by type:
  `$ declare -a|A|f|i|r|x  # choose one option`

- Example: `$ declare -rx pi=3.1415927`

bash Array Variables

- Define indexed arrays:
  $ declare -a flower='([0]="rose" [1]="daisy" [2]="violet")'
  $ flower=(rose daisy violet) # Alt. Def.
  $ echo ${flower[*]}   # to display values

- Define associative arrays:
  $ declare -A fish='([smelt]="3" [salmon]="6" [tuna]="8")'
  $ echo ${fish[*]}   # to display values

5. Functions

name() function name()

bash Functions

- 3 ways to define:
  ```
  > name() { command; ...; return; }
  > function name { command; ...; return; }
  > name()
  {
  command
  ...
  return
  }
  ```
- Functions and calling programs share the same shell

bash Functions (2) U1

- Function names also share variable name space
- Define in memory on commandline; evaluate (run) by invoking name as a command
- Save in a file and define in memory via
  $ . ./functionfile   # reuse between login sessions
- $ export -f functionname # reuse for future shells
- Show Functions (typeset obsolete):
  $ declare -F   # show [declare -f] names only
  $ declare -f   # shows names and definitions
- Remove Function
  $ unset -f name

Function Examples

- Directory: mcd() { mkdir -p $1 ; cd $1; }
- Selective Lists: lsext() { find . -type f -iname "*.'${1}' ' -exec ls -l {} \; ; }
- Create random password: rpass() { cat /dev/random | tr -cd '[:graph:]' | head -c ${1:-12}; echo; }
- Get IP address of a given interface: getip() { /sbin/ifconfig ${1:-eth0} | awk '/inet addr/ {print $2}' | awk -F: '{print $2}' ; }

Function Examples (2)

- Surveillance function

```
wait_for_user()
# wait for a user to log in on this system
# usage: wait_for_user userid repeattime
until who | grep “$1” > /dev/null
do
sleep ${2:-30}   # default time=30 seconds
done
return
}
```

6. Manipulating Commands

history fc -l <esc>- r

bash Command History

- bash History: maintains a list of recently issued commandlines (events) that offers a quick way to repeat or edit and rerun past commands.
- Your Command History stored in file: .bash_history
- Advantages:
  - keeps a recent record of your session
  - lets you (modify and) rerun past commands
  - lets you review commands having errors

bash Command History (2)

- History Variables:
  HISTSIZE=1000 Maximum No. of events saved during a login session
  HISTFILE=~/.bash_history History file path
  HISTFILESIZE=1000 Maximum No. of events saved between login sessions

- Display history file contents:
```
$ history [start [end]]  # or run fc -l
```

- Edit command(s) in history file contents and run result:
```
$ fc [start [end]]  # vim editor default else use fc -e vim
```

- Repeat last command:
```
$ r [pattern=replacement] [command|event No.]
# an alias for running fc -s; can also type !!
```

33

bash Command History (3)

- **Command-Line Event Desig-nators**

Designator	Meaning
!	Starts a history event
!!	previous command
!n	Command No. n in history
!-n	The nth preceding command
!string	Most recent command starting with “string”
!?string[?]	Most recent command containing “string”
!#	Repeat current command typed so far
!{event}	Isolate event designator

bash Command History (4)

- Argument Word Designators

Designator	Meaning
n	Nth word; word 0 = command
^	First word = 1st argument
$	Last word (argument)
m-n	All word in range word m through word n; missing m means 0
n*	all words from word n to end of line
*	all words but command name (=1*)
%	word matched by most recent ?string?
^pat^rep^	short for [g]s/old/new/

7. bash Option Behavior

bash Options

- Login bash shell is called with certain options. Use **$-** to view current option letters:
  $ echo $-   # h=hash commands, i=interactive shell, m=job control on B=brace expansion H=history expansion.
  himBH

- To enable commandline editing, type: **set -o vi**

- See **set** options <gnu.org/software/bash/manual/html_node/The-Set-Builtin.html> and **shopt** options <gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html>

bash Settings

- To show global variable names and values, use:
  ```
  $ [print]env | less
  ```
- To show or modify global variable values in the current shell or for a subshell, use:
  ```
  $ env [-i|-u name] [-] [name=value]...[commandline]
  ```
- Example:
  ```
  $ cat display_xx
  echo "Running $0"
  echo $xx
  $ env xx=process ./display_xx   # Alt.: xx=process ./display_xx
  Running ./display_xx
  process
  ```

8. Reading, Writing, Modifying Strings

expr

bash String Manipulation

- String Length variations:
  - $ echo ${#string}
  - $ expr length $string
  - $ expr "$string" : '.*'
  - $ echo $string | expr $(wc -c) - 1

- expr built-in form-string manipulation:

  ```
  expr STRING REGEXP
  expr match STRING REGEXP
  expr substr STRING POS LENGTH # POS is 1-based
  expr index STRING CHARS # 0 if no CHARS found
  expr length STRING
  ```

9. bash Expansions

arithmetic relational command substitution brace substitution

bash Filename Expansion

- **File name Expansion (wildcards)**

Symbol	Meaning	Example
?	Represents any single character	echo ? a?a
*	Represents zero or more characters	ls * ls *.txt
[ ], [!] [[:class:]]	Represents a list or range of characters (! means not)	ls [aeiou]* ls *.??[a-z0-9]
{ }	alternatives list	cp {*.doc,*.pdf} ~ echo a{b,c}d

Extended Filename Expansion

- **Extended Pattern Matching**

Symbol	Meaning	Example
?(pat1|... |patn)	0 or 1 of a pattern collection (+ null)	$ ls ?(x|y1) x
@(pat1| ...|patn)	Exactly 1 of a pattern out of n	$ ls @(x|y1) x
*(pat1| ...|patn)	0 or more of a pattern collection	$ ls *(x|y1) x xx xxx xxxx
+(pat1| ...|patn)	1 or more of a pattern collection	$ ls +(x|y1) x xx xxx xxxx
!(pat1| ...|patn)	Any pattern except these	$ ls !(z1|y1) x xx xxx xxxx

Arithmetic Operators

- Used in **expr** and **let** [same as] **(( ))**
- Symbols: {+, -, *, /, %, **, =, +=, -=, *=, /=, %=, <<, <<=, >>, >>=, &, $=, |, |=, ~, ^, ^=, !, &&, ||, ‘,’ }
- See <tldp.org/LDPabs/html/ops.html>

bash Numeric Constants

- bash exclusively uses integer arithmetic, not decimal numbers
- Recognizes Octal numbers (Leading 0), Hexadecimal numbers (Leading 0x), other BASE#NUMBER (2 ≤ BASE ≤ 64) ([01] ≤ NUMBER ≤ [0-9a-zA-Z@_])
- See <tldp.org/LDP/abs/html/numerical-constants.html>

bash (( )) Construct

- Provides arithmetic expansion and evaluation
- " = " permitted inside (( ))
  "$" not required inside (( ))
- Relational operators (<=, >=, <, >, ==, !=)
- Pre and Post variable Increment ++ --
  ```
  $ a=1; echo $(( ++a*2 )) # 4
  $ a=1; echo $(( (a*2)++ )) # 3
  $ a=1; echo $(( --a*2 )) # 0
  $ a=1; echo $(( (a*2) -- )) # 1
  ```
- ? : trinary operator `$ a=2; echo $(( t = a>0?1:-1 )) # result =1`
- See <tldp.org/LDP/abs/html/dblparens.html>

Operator Precedence U1

- Arithmetic and Relational Expressions are evaluated using precedence order (e.g. Please Excuse My Dear Aunt Sally mnemonic standing for: Parenthesis, then exponents, then multiplication or division, then addition or subtraction)
- $ echo $((5+3*4)) # Result=17, not 32
- See Table <tdlp.org/LDP/abs/html/opprecedence.html>

Comparison Operators

- Use [ ] or [[ ]] to compare strings; Use (( )) to compare numbers
- The result or status of any Linux command is: 0 means successful; non-zero means unsuccessful
- View the result via $ **echo $?** immediately after the linux command But: ((n)) is successful if n ≠ 0, unsuccessful if n = 0

Comparison Operators (2)

- **test and [ ]**

Symbol	Meaning: true status if
n1 **-eq** n2	two numbers are equal
n1 **-ne** n2	two numbers are not the same
n1 **-gt** n2	n1 is bigger than n2
n1 **-lt** n2	n1 is less than n2
n1 **-ge** n2	n1 is at least as big as n2
n1 **-le** n2	n1 is at most as big as n2
!	not
-a	Boolean AND
-o	Boolean OR
-z s1	string length is 0
-n s1	string length more than 0
s1 **=** s2	both strings are identical
s1 **!=** s2	each string is different than the other
s1	string is not the null string (empty)

Comparison Operators (3)

- [[ ]] Comp-arisons

Symbol	Meaning: true status if
s1 = s2	both strings are identical
s1 = w.c.pattern	strings matches wild card attern
s1 != s2	each string is different than the other
s1 != w.c.pattern	string doesn't match wild card pattern
s1 > s2	s1 follows s2 in alphabetical order
s1 < s2	s1 precedes s2 in alphabetical order
-z s1	string length is 0 (null string)
-n s1	string length larger than 0

Command Substitution

- Uses Linux to produce commandline ingredients
- Form: $(command) {equivalent to ` command `}
- Command Substitutions may be nested to arbitrary levels since '(' different than ')'. They always start a subshell
- Example:

```
$ echo Today\'s date and time are $(date).
Today's date and time are Fri Aug 8 08:32:19 PDT 2015.
```

- See <tldp.org/LDP/abs/html/commandsub.html>

Process Substitution U1

- **Process Substitution** sends the output of one or more processes to the stdin of another process.
- Form: A command list is enclosed in parentheses:
  >(command_list) # ; separator for list items -- stdout
  <(command_list) # stdin
- /dev/fd/<n> is used to transfer stdout to stdin. No subshell is started with this kind of substitution.
- Examples:

```
$ wc <(cat bashman)    # lines, words, chars
   7748  42256  314136  /dev/fd/63
$ wc <(cat bashman; echo today)
   7749  42257  314142  /dev/fd/63
$ diff <(ls $firstdir) <(ls $seconddir) # compare 2 dirs.
$ comm <(ls -l) <(ls -al) # compare options output
```

- See <tldp.org/LDP/abs/html/process-sub.html>

52

Brace Expansion U1

- Forms: {a, b, c} # smallest list is: {,}
  {1..10} or {a..z} or {M..A} # 1st 10 integers, all letters, reversed order of letters
- Brace Expansions may be nested. Strings are produced, not filenames.
  ```
  $ echo a{A{1,2},B{3,4}}b
  aA1b aA2b aB3b aB4b
  ```
- See <linuxcommand.org/lc3_lts0080.php>

Numerical Calculation

expr let (( )) bc dc awk

Integer Arithmetic

Symbol	expr	let	(( ))
+	expr 3 + 5	let R="3 + 5"	((R = 3 + 5))
-	expr 5 - 3	let R=5-3	((R=5-3))
*	expr 3 * 5	let R='3 * 5'	((R = 3 * 5))
/	expr 5 / 3	let R="5/3"	((R=5/3))
%	expr 5 % 3	let R="5%3"	((R=5%3))
**	NA	let R="3**5"	((R=3**5))
++, --	NA	let R=++var; let S=var--	((R = var++)); ((S = --var))
+=, -=, *=, /=	NA	let R=var+=1; let S=var-=2	((R = var*=3)); ((S = var/=4))

Decimal Arithmetic

- **bc (1)** uses decimal arithmetic with arbitrary precision results on the command line or interactively.
- Arithmetic symbols are the same except for ^ replacing ** for exponentiation
- Standard functions are: scale, length, read and sqrt (=n^1/2; use fractional exponents for higher roots)
- Use:
```
$ echo “scale=2; 3*17.5” | bc  # or echo “3*17.5” | bc -l
        52.5
```
- Use:
```
bc -l <<< “3.4+7.0/8.0-(5.94*3.14)”    # Here string example
-14.37660000000000000000
```
- Add **calc()** function to ~/.bashrc :
```
calc(){ printf “%.2f\n” $(echo “$@” | bc -l); }  # 2 place rounding
$ calc 2+3*8/7   # means: 2 + (3*8)/7
5.43
```
- See <shell-tips.com/2010/06/14/performing-math-calculation-in-bash/>

Decimal Arithmetic (2)

- **dc (1)** a "reverse polish" desk calculator used by **bc** or scripts rather than humans.
- unary minus sign is an underscore.
- Example: sqrt[((1234*2)-468)/2] to 10 places

```
$ dc <<< "1234 2 * 468 - 2 / 10 k v p"
31.6227766016
```

- See <computerhope.com/unix/udc.htm>

Decimal Arithmetic (3)

- **awk (1)** offers C-like arithmetic operators to evaluate expressions in its ‘pattern’ and/or {action sequences }
- Examples:
  ```
  $ awk ‘NR % 2 == 0’ /etc/passwd #shows even numbered lines
  $ awk ‘END {printf “%5.10f”, sqrt(((1234*2)-468)/2); }’ anyfilename
  31.6227766017
  ```
- See <funtoo.org/Awk_by_Example_Part_1>

bash Control Flow Commands

if then else for while until do done case esac select

if then else command

- Command-lists can't be empty
- Forms:
  if command-list; **then** command-list1;[**elif** command-list2; **then** command-list3; **else** command-list4;] **fi**

-
  ```
  if command-list
  then
     command-list1
  [ elif
     command-list2
  then
     command-list3
  ...
  else
     command-list4 ]
  fi
  ```

Conditional Logic Example

- Example:

```
$ U=userid
$ if who | grep “$U” > /dev/null
 then echo Your friend $U is logged in
 else echo We are $U-free.
 fi
```

bash Loop Commands

- for [in] do done command executes a commandlist in the body of the loop repeatedly, in order to process a series of string values contained in a list of items.

- Forms:

```
for variablename [in listofitems | or contents of $@ ]
do
    commandlist
 done
```

- Example:

```
$ for i in {1..10..2}; do; echo Hello $i times; done
# produces 5 lines of Hello {1,3,5,7,9} times.
```

bash Loop Comands (2)

- **{while,until} do done** command continues to run commandlist2 as long as the commandlist1 is {true (0 status), false (1 status) }

- Form: **while|until** commandlist1

```
do
   commandlist2
done
```

- Examples: Infinite or Event loop:

```
$ while (( 1 )); do echo still looping; done
$ until (( 0 )); do echo still looping; done
```

  Monitor i's value in a loop:

```
$ i=1; while (( i <= 10)) do; echo i is $i; (( i++ )); done
```

bash Loop Examples

-
  ```
  $ set apple banana cherry
  $ while [ $# -gt 0 ]; do echo $1; shift;
  done
  ```

-
  ```
  $ lookfor=<userid>
  $ until who|grep “^$lookfor” >
  /dev/null; do sleep 60; done
  $ echo $lookfor has logged on at $(date)
  $ who
  ```

case Decision Command

- case in esac Chooses a commandlist based on evaluation of an expression rather than the status of a commandlist.

- Form: case expression in
  case1) commandlist1 ;;
  case2) commandlist2 ;;
  *) default commandlistn ;;
  esac

- expression, case1, case2, etc are usually strings or variable values

case Example

- **Initialization script code segment:**

```
USAGE=”Usage: $0 {start|stop|restart|condrestart|
status}”
case “$1” in
     start) app start ;;
      stop) app stop ;;
 restart) app stop; app start ;;
 condrestart) if [ “x$(pidof app)” != x ]
                        then stop app; start app; fi ;;
           *) echo $USAGE; exit 1 ;;
 esac
```

bash Shell Scripts

#! $USAGE exit n bash -vx

bash Scripts

- Definition: A text file containing a series of Linux commands to be executed within the context of a bash shell.
- Each line in a script file is a single command except when last character is \ or **<< word** or command name is part of a multiline command.
- Comments begin after # and go to the end of the line
- Line 1 of script: **#! /bin/bash [-oneoption]** directs the Kernel to use the bash program to interpret this script.
- A **USAGE** line defines the variable **USAGE** to show the script name and proper usage. e.g.:
  USAGE=”Usage: myscript.bash file1 file2”
- See bash resource: **<tldp.org/LDP/abs/html/index.html>**

bash Script Template

```
#! /bin/bash
USAGE=”Usage: template.bash”
# Program name: template.bash
# Author: Robert Katz
# Date: August 7, 2015
# Purpose: A template for your scripts
# Your actual commands go below this line

# END OF template.bash ( last line of
script )
```

bash Script Exercise

- 1. Write a shell script (program) named **diet** that displays any file without the first and last **n** lines, where **n** is an integer. Use the following syntax:
  $ **diet -n file**
  Type in the program and test it out.
- 2. Rewrite the **diet** script as a function.

bash Script Exercise Answer (1)

- #! /bin/bash
  USAGE=”Usage: diet -number filename”
  # Program name: diet
  # Author: Robert Katz
  # Date: 8/3/2015
  # Purpose: To strip off lines from the top and the bottom of any text file.
  ### Note: For a function, replace ‘exit’ with ‘return’ everywhere in the script

bash script Exercise Answer (2)

```
# 1. test that there are 2 arguments
if [ $# -ne 2 ]
then
    echo $USAGE; exit 1
fi

# 2. Store 1st argument in N as an
# integer and strip off the leading ‘-’
declare -i N=”${1#-}”
```

bsh script Exercise Answer (3)

```
# 3. store 2nd argument in FILE and
# verify that it exists.
FILE=”$2”
if [ ! -f $FILE ]
then
   echo “File not found”; echo $USAGE
   exit 2
fi
```

bsh script Exercise Answer (4)

```
# 4. Ready to Process. Determine size
# of FILE
# How many lines in the entire file?
COUNT=$(cat $FILE | wc -l)

# Last line number to output using head
LAST=$(( $COUNT - $N ))

# Number of lines to output using tail
FIRST=$(( $LAST - $N ))
```

bsh script Exercise Answer (5)

```
# 5. Verify that the file is big enough or
# do not output anything.
if [ $FIRST -gt 0 ]
then
  head -$LAST $FILE | tail -$FIRST
fi
exit
# END OF diet
```

traps and signals

- **trap** builtin: In a script, **trap** changes the way signals are handled from default script termination. The signal list is produced by **kill -l** .
- **trap** is set for any signal (not sigkill=9), which ignores all traps for it
- Forms: **trap** # Lists traps set in current shell
  trap " " signal(s) # Ignore listed signals
  trap - signal(s) # Restore default processing for listed signals
  trap 'action' signal(s) # Trigger the action to run if signal(s) received
- traps may also be set for 3 fake signals:
  EXIT trigger trap action when successful exit occurs
  ERR trigger trap action whenever a command has a non-zero status
  DEBUG trigger trap action after every command
- Subshells inherit trap commands only to ignore or restore default handling, no customized action.

trap Example

```
$ cat trap-1.bash
#! /bin/bash
USAGE=”Usage: trap-1.bash”
# setting traps on INT and QUIT signals
declare -ix c; declare -ix rt; (( c = rt = 0 ))
trap “echo Received INT signal c=$c” SIGINT
trap “echo Received QUIT signal rt=$rt” SIGQUIT
while (( c < 1000000 )); do (( c++));((rt++)); done
echo “The final answer is $rt”; exit
#END OF trap-1.bash
$ ./trap-1.bash &
[1] 12345
$ kill -INT %1  # or kill -2 12345 or <Ctrl-C>
Received INT signal; c = 1578
$ kill -QUIT %1 # or kill -3 12345
Received QUIT signal: rt = 17931066
The final answer is 500000500000
```