CS29206 Systems Programming Laboratory
Spring 2024

Introduction to bash

Abhijit Das
Pralay Mitra

Modified in Autumn 2025 by CS29206 Instructor: Mostly formatting changes, deleting of a few slides, and few minor changes

What is a Unix shell?

- A shell is a command interpreter.
- It can run interactively or non-interactively.
- A shell can be programmed like a high-level programming language.
- Some common Unix shells

sh	The original Bourne shell written by Steve Bourne of AT&T Bell Labs.
bash	The Bourne Again Shell is an extension of the original Bourne shell.
ksh	The Korn shell written by David Corn is another extension of the original Bourne shell.
csh	The C Shell is developed for Berkeley Unix.
tcsh	An extension of csh (the T comes from TENEX and TOPS-20 OS)
rbash, rksh	Restricted shells

- Different shells have different syntaxes. We will use bash.

The default shell

- Called the login shell, written in /etc/passwd
- Most Linux versions supply bash as the login shell

```
$ echo $SHELL
/bin/bash
```

Run a set of commands in non-interactive mode

```
$echo  $SHLVL
1
$ bash  -c  'cal March 2023'
March  2023
Su  Mo  Tu  We  Th  Fr  Sa
 1   2   3   4
 5   6   7   8   9  10  11
12  13  14  15  16  17  18
19  20  21  22  23  24  25
26  27  28  29  30  31
```

Environment variables

- The shell starts with a set of default variables called **environment variables**.
- In a non-interactive shell, these variables are stored in BASH_ENV.
- In an interactive shell, use set to see all the defined variables.

```
$ set
BASH=/bin/bash
COLUMNS=100
GROUP=student
HOME=/home/foobar
HOSTNAME=FBserver
LANG=en_US.UTF-8
LINES=25
LOGNAME=foobar
OSTYPE=linux
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/home/foobar/bin:.
PS1='$ '
PS2='> '
PWD=/home/foobar
SHELL=/bin/bash
SHLVL=1
TERM=vt100
UID=1000
USER=foobar
...
$
```

User-defined variables

- New variables can be defined by the user.
- The naming conventions are similar to as in C.
- Define a new variable as VAR=VALUE
- Spaces are not allowed before or after =.
- The value of the variable VAR is accessed as $VAR or as ${VAR}.
- A variable can be undefined by unset VAR.

```
$ MY_NAME=Foolan
$ MY_FULL_NAME=Foolan Barik
Barik: command not found
$ MY_FULL_NAME="Foolan Barik"
$ echo $MY_NAME
Foolan
$ echo $MY_NAME $MY_FULL_NAME
Foolan Foolan Barik
$ echo $MY_NAME; echo $MY_FULL_NAME
Foolan
Foolan Barik
$ unset MY_NAME
$ echo $MY_NAME
```

Three types of quotes

- Double quotes expand the variable values specified by $VAR.
- Use \$ within double quotes to take $ literally.
- Single forward quotes take $ literally, and do not expand variable values.
- Single backward quotes execute the command after variable substitution (if any).

```
$ MYNAME="Foolan Barik"
$ echo "Welcome MYNAME"
Welcome MYNAME
$ echo "Welcome $MYNAME"
Welcome Foolan Barik
$ echo "Welcome \$MYNAME"
Welcome $MYNAME
$ echo ’Welcome $MYNAME’
Welcome $MYNAME
$ echo ’Welcome \$MYNAME’
Welcome \$MYNAME
$ echo ‘Welcome $MYNAME‘
Welcome: command not found
```

Examples of running commands by back-quoting

```
$ echo ‘ls /‘
bin boot cdrom dev etc home lib lib32 lib64 libx32 lost+found media mnt opt proc root run sbin snap srv sys
tmp usr var
$ ‘echo $HOME‘
bash: /home/foobar: Is a directory
$ echo ‘echo $HOME‘
/home/foobar
$ ls ‘echo $HOME‘ | wc
     56      56     639
$ ls ‘echo $HOME‘/spl
asgn/  book/  books.txt  Format.docx  man/  prog/  slides/  syllabus.txt  tmp/
$
```

Note: Instead of back quotes, you can use `$(...)`.

```
$ echo $(ls /)
bin boot cdrom dev etc home lib lib32 lib64 libx32 lost+found media mnt opt proc root run sbin snap srv sys
tmp usr var
$ $(echo $HOME)
bash: /home/foobar: Is a directory
$ echo $(echo $HOME)
/home/foobar
$ ls $(echo $HOME) | wc
     56      56     639
$ ls $(echo $HOME)/spl
asgn/  book/  books.txt  Format.docx  man/  prog/  slides/  syllabus.txt  tmp/
$
```

Exporting user-defined variables

- You need to export a variable if you want to continue to access those variables in sub-shells.
- Exporting can be done separately after defining or at the time of defining.

```
$ MYNAME=Foolan
$ export MYNAME
$ export MY_NAME=Foolan
$ MY_FULL_NAME="Foolan Barik"
$ bash
$ echo $SHLVL
2
$ echo $MYNAME
Foolan
$ echo $MY_NAME
Foolan
$ echo $MY_FULL_NAME
```

Special variables

- The command-line parameters are called positional parameters.
- These can be accessed inside shell scripts or functions.

$* or $@	All the command-line parameters in a single strings
$#	The number of command-line parameters (excluding the command)
$0	The command
$1, $2, . . .	The first, second, . . . command-line parameters
$?	Exit status of the last command (0 means successful termination, non-zero means unsuccessful termination)

```
$ ls/
bash: ls/: No such file or directory
$ echo $?
127
$ ls /
bin   cdrom  etc   lib    lib64   lost+found  mnt  proc  run   snap  sys  usr
boot  dev    home  lib32  libx32  media       opt  root  sbin  srv   tmp  var
$ echo $?
0
$
```

Example of positional parameters

```
$ parameters () {
> echo "\$0 = $0"
> echo "\$# = $#"
> echo "\$* = $*"
> echo "First parameter: $1"
> echo "Second parameter: $2"
> echo "Third parameter: $3"
> }
```

```
$ parameters a b c d e
$0 = bash
$# = 5
$* = a b c d e
First parameter: a
Second parameter: b
Third parameter: c
$ parameters foolan barik
$0 = bash
$# = 2
$* = foolan barik
First parameter: foolan
Second parameter: barik
Third parameter:
```

Reading variables

- You can read one or more variables from the shell.

```
$ echo -n "Enter your name: "; read MYNAME
Enter your name: Foolan Barik
$ echo $MYNAME
Foolan Barik
$ echo -n "Enter your name: "; read FIRSTNAME LASTNAME
Enter your name: Foolan Kumar Barik
$ echo $FIRSTNAME
Foolan
$ echo $LASTNAME
Kumar Barik
$ read x y
5
$ echo "x = $x, y = $y"
x = 5, y =
```

Read-only variables

- Make a variable read-only by declare -r VAR.
- Subsequent changes in VAR are no longer possible.
- Some default shell variables are read-only.

```
$ MYNAME="Foolan Barik"
$ declare -r MYNAME
$ MYNAME="Foolan Kumar Barik"
bash: MYNAME: readonly variable
$ declare -r SHORTNAME="F. Barik"
```

```
$ declare -r
declare -ar BASH_VERSINFO=([0]="5" [1]="0"
[2]="17" [3]="1" [4]="release" [5]="x86_64-pc-
linux-gnu")
declare -ir EUID="1000"
declare -r MYNAME="Foolan Barik"
declare -ir PPID="9136"
declare -r SHORTNAME="F. Barik"
declare -ir UID="1000"
$ read UID
1234
bash: UID: readonly variable
```

Only some read-only shell variables shown

String operations

- **Length of a string:** `${#S}`
- **Substring from index *i* to end:** `${S:$i}`
- **Substring from beginning to index *i*:** `${S: -$i}` (space needed after `:`)
- **Substring of length *j* starting from index *i*:** `${S:$i:$j}`
- **Substring from index *i* from the beginning to index *j* from the end:** `${S:$i:-$j}`
- **Contatenating strings:** `S="$S1$S2$S3..."` (no space between the components)
- **Inserting substring at index *i*:** `S="${S:0:$i}$T${S:$i}"`
- **Deleting substring from index *i* to index $j - 1$:** `S="${S:0:$i}${S:$j}"`

Examples of string operations

```
$ S="abcdefgh"
$ T="ghijklmnop"
$ S="$S$T"
$ echo "$S has length ${#S}"
abcdefghghijklmnop has length 18
$ S="${S:0:8}${S:10}"
$ echo "$S has length ${#S}"
abcdefghijklmnop has length 16
```

```
$ echo ${S:4}
efghijklmnop
$ echo ${S: -4}
mnop
$ echo ${S:4:4}
efgh
$ echo ${S:4:-4}
efghijkl
$
```

Array variables

- Arrays can be declared using `declare -a ARRNAME`.
- There is no limit on the array size.
- Array indexing is zero-based.
- Array elements can be set as `ARRNAME[IDX]=VALUE`.
- Array entries can be accessed as `${ARRNAME[IDX]}`.
- All array entries can be listed as `${ARRNAME[@]}` or `${ARRNAME[*]}`.
- All array indices can be listed as `${!ARRNAME[@]}` or `${!ARRNAME[*]}`.
- The array size is obtained as `${#ARRNAME[@]}` or `${#ARRNAME[*]}`.
- A read-only array can be assigned entries only during declaration.
- No entry of a read-only array can be changed (even if undefined during declaration).
- A read-only array (or variable) cannot be unset.

Examples of arrays

```
$ declare -a MYARR
$ MYARR[0]="zero"; MYARR[1]="one";
MYARR[2]="tw0"; MYARR[4]="four"
$ MYARR[2]="two"
$ MYARR[5]="five"
$ echo "${MYARR[0]}, ${MYARR[1]},
${MYARR[2]}, ${MYARR[3]}, ${MYARR[5]}"
zero, one, two, , five
$ echo ${MYARR[@]}
zero one two four five
$ echo ${!MYARR[@]}
0 1 2 4 5
$ declare -iar FIB=([0]=0 [1]=1 [2]=1 [3]=2 [4]=3
[5]=5 [6]=8 [7]=13 [8]=21 [9]=34)
```

```
$ echo ${FIB[5]}
5
$ echo ${FIB[*]}
0 1 1 2 3 5 8 13 21 34
$ echo ${!FIB[*]}
0 1 2 3 4 5 6 7 8 9
$ echo ${FIB[10]}

$ FIB[10]=55
bash: FIB: readonly variable
$ unset MYARR
$ echo ${MYARR[0]}
$ unset FIB
bash: unset: FIB: cannot unset: readonly
variable
```

Operations on arrays

- **Quick initialization:** `ARR=(elt0 elt1 elt2 elt3 ...)`
- **Appending:** `ARR+=(new1 new2 new3 ...)`
- **Accessing subarrays**
  - From index i to end: `${ARR[@]:$i}`
  - *j* elements starting from index *i*: `${ARR[@]:$i:$j}`
- **Inserting elements at index *i*:**
  `ARR=(${ARR[@]:0:$i} new1 new2 new3 ... ${ARR[@]:$i})`
- **Deleting elements:** `unset ARR[$i1] ARR[$i2] ...`
- **Compact indexing after deletion:** `ARR=(${ARR[@]})`
- **Concatenating two (or more) arrays:** `ARR=(${ARR1[@]} ${ARR2[@]} ...)`

Examples of array manipulation

```
$ P=(2 3 5 7)
$ echo ${P[@]}
2 3 5 7
$ P+=(11 13 17 19 31 37)
$ echo ${P[@]}
2 3 5 7 11 13 17 19 31 37
$ P=(${P[@]:0:8} 21 23 29 ${P[@]:8})
$ echo ${P[@]}
2 3 5 7 11 13 17 19 21 23 29 31 37
$ unset P[8]
$ echo ${P[@]}
2 3 5 7 11 13 17 19 23 29 31 37
$ echo ${!P[@]}
0 1 2 3 4 5 6 7 9 10 11 12
```

```
$ P=(${P[@]})
$ echo ${P[@]}
2 3 5 7 11 13 17 19 23 29 31 37
$ echo ${!P[@]}
0 1 2 3 4 5 6 7 8 9 10 11
$ Q=(41 43 47)
$ P=(${P[@]} ${Q[@]})
$ echo ${P[@]}
2 3 5 7 11 13 17 19 23 29 31 37 41 43
47
$
```

Associative arrays (Hashes)

- Associative arrays are declared using `declare -A HNAME`.
- Associative arrays are indexed by strings (integer indices are converted to strings).
- Setting entries: `HNAME[STR]=VALUE`.
- Accessing entries: `${HNAME[STR]}`.

```
$ declare -A MYINFO=(["name"]="Foolan barik" ["fname"]="Foolan" ["lname"]="Barik")
$ MYINFO["cgpa"]="9.87"
$ MYINFO["height"]="5'08''"
$ MYINFO["mobile games"]="Numberlink:Slitherlink:Sudoku:2048"
$echo "\"${MYINFO[fname]} ${MYINFO[lname]}\" likes games ${MYINFO[mobile games]}"
"Foolan Barik" likes games Numberlink:Slitherlink:Sudoku:2048
$ echo ${MYINFO[@]}
Foolan Numberlink:Slitherlink:Sudoku:2048 5'08'' Barik Foolan barik 9.87
$ echo ${!MYINFO[@]}
fname mobile games height lname name cgpa
$ for key in ${!MYINFO[@]}; do echo $key -\> ${MYINFO[$key]}; done
fname -> Foolan
mobile ->
games ->
height -> 5'08''
lname -> Barik
name -> Foolan barik
cgpa -> 9.87
$
```

Arithmetic expressions

- Use the syntax `$((EXPRESSION))`.
- This works only with integer variables.
- Strings are automatically converted to integers.
- Non-numeric strings and undefined values are converted to 0.
- Standard integer operators work as in C.
- `**` is the exponentiation operator.
- `$` may be omitted for accessing variables.

Examples of arithmetic expressions

```
$ a=3; b=4; c=-5
$ echo $(($a + $b * $c - 6))
-23
$ echo $((a + b * c - 6))
-23
$ z=$((a ** 2 + b ** 2))
$ echo $z
25
$ echo $((z / y))
bash: z / y: division by 0 (error token is "y")
$ y="Non-numeric"
$ echo $((z / y))
bash: z / y: division by 0 (error token is "y")
```

```
$ declare -a FIB=([0]=0 [1]=1)
$ n=2; FIB[$n]=$((FIB[n-1]+FIB[n-2]))
$ n=3; FIB[$n]=$((FIB[n-1]+FIB[n-2]))
$ n=4; FIB[$n]=$((FIB[n-1]+FIB[n-2]))
$ n=5; FIB[$n]=$((FIB[n-1]+FIB[n-2]))
$ n=6; FIB[$n]=$((FIB[n-1]+FIB[n-2]))
$ echo ${FIB[@]}
0 1 1 2 3 5 8
$ echo ${!FIB[@]}
0 1 2 3 4 5 6
$
```

Floating-point calculations

- Use the arbitrary-precision calculator `bc`.
- The default precision is 0.
- Set `scale` to define the precision in decimal digits.

```
$ num=22; den=7
$ approxpi=‘echo "$num / $den" | bc‘
$ echo $approxpi
3
$ approxpi=‘echo "scale = 10; $num / $den" | bc‘
$ echo $approxpi
3.1428571428
$ num=355; den=113; echo ‘echo "scale = 10; $num / $den" | bc‘
3.1415929203
$
```

Functions

- A function can be defined as

```
function FNAME () {
    commands
}
```

- The keyword `function` before `FNAME` is optional.
- After the definition, `FNAME` behaves like a command.
- The positional parameters `$*`, `$#`, `$1`, `$2`, . . . refer to the command-line arguments.
- Use `declare -f` to see a listing of all defined functions.
- A function can be undefined by `unset FNAME`.
- A function can be recursive.
- The shell variable `FUNCNEST` can be set to a positive integer to limit the recursion depth.

Examples of a simple function

```
$ function twopower () {
> echo "Usage: twopower exponent"
> echo "2 to the power $1 is $((2 ** $1))"
> }
$ twopower 10
Usage: twopower exponent
2 to the power 10 is 1024
$ twopower 30
Usage: twopower exponent
2 to the power 30 is 1073741824
$ twopower 60
Usage: twopower exponent
2 to the power 60 is 1152921504606846976
```

```
$ twopower 100
Usage: twopower exponent
2 to the power 100 is 0
$ twopower
Usage: twopower exponent
bash: 2 ** : syntax error: operand
expected (error token is "** ")
$ twopower -3
Usage: twopower exponent
bash: 2 ** -3: exponent less than 0
(error token is "3")
$
```

Return a value or not?

- Only an unsigned 8-bit value can be returned.
- Like other commands, the return value is treated as an indicator of successful completion.
- Set a non-local variable instead if you want to return a value (string or integer).

```
$ function twopower () { return $((2 ** $1)); }
$ twopower 2; retval=$?; echo $retval
4
$ twopower 7; retval=$?; echo $retval
128
$ twopower 8; retval=$?; echo $retval
0
$ function twopower () { retval=$((2 ** $1)); }
$ twopower 8; echo $retval
256
$ twopower 50; echo $retval
1125899906842624
$
```

Scope of variables

- Declare local variables using the keyword local.
- A local variable shadows a variable with the same name in the outer scope.
- A nested function call sends global and local variables to the called functions.
- The innermost scope where a variable is defined is used.

```
$ x=3; y=4; z=5
$ fx () { local x=6; echo "x = $x, y = $y, z = $z, w = $w"; }
$ fx
x = 6, y = 4, z = 5, w =
$ fxy () { local y=7; local w=8; local x=9; fx; }
$ fxy
x = 6, y = 7, z = 5, w = 8
$ fx
x = 6, y = 4, z = 5, w =
$ fxyw () { local y=7; w=8; fx; }
$ fxyw
x = 6, y = 7, z = 5, w = 8
$ fx
x = 6, y = 4, z = 5, w = 8
$ echo "x = $x, y = $y, z = $z, w = $w"
x = 3, y = 4, z = 5, w = 8
```

Bash commands

- A binary executable like a.out, echo, firefox, grep, or xterm.
- A script file (for sed, gawk, or bash).
- A bash function behaves like a command.
- There are some built-in commands (like cd) that only the shell understands. No executable files exist for these commands.
- A command takes zero or more command-line arguments.
- Upon completion, a command returns a status.
- A command runs in the background:

```
$ cmd arg1 arg2 ... &
```

- The file descriptors for a command can be redirected using `<`, `>`, `2>`, `>>`, `2>>`, `|`, `2|`.

Wild cards

- Bash has limited ability to handle regular expressions in command-line arguments.
- Bash substitutes all matches one after another in the command line.
- Quoting (single or double) prevents this substitution.
- Three types of patterns:

`*`	Match any string
`?`	Match a single character
`[...]`	Match a single character in a range

- The range may be
  - A range of letters specified by `-`, like `a-g` or `0-5`.
  - A special range specified as `[:SPLRNG:]`, where `SPLRNG` can be `alpha`, `digit`, `alnum`, `upper`, `lower`, `blank`, `space`, `xdigit`, and so on.

Wild card examples

- `*.txt` matches any file with extension `.txt`.
- `.*` matches all hidden files (and directories).
- `?.txt` matches any file with a single-letter name and with an extension `.txt`.
- `???*.txt` matches any file with name having at least three characters and with an extension of `.txt`.
- `[0-9]*` matches any file starting with a digit.
- `[[:alpha:]][[:digit:]]*.jpg` matches any jpeg file whose name starts with an alphabetic character followed by a digit followed by any string.
- `spl/progs/*.c` matches all C source files in the sub-sub-directory `spl/prog/`.

Wild card uses

```
$ ls -p spl/
asgn/  book/  books.txt  Format.docx  man/  prog/  slides/  syllabus.txt  tmp/
$ ls -p spl/*.txt
spl/books.txt  spl/syllabus.txt
$ ls -p spl/[[:lower:]]*.*
spl/books.txt  spl/syllabus.txt
$ ls -p "spl/*.txt"
ls: cannot access 'spl/*.txt': No such file or directory
$ spltext=spl/*.txt
$ ls $spltext
spl/books.txt  spl/syllabus.txt
$ ls "$spltext"
ls: cannot access 'spl/*.txt': No such file or directory
$ alltext=*.txt
$ ls spl/$alltext
spl/books.txt  spl/syllabus.txt
$ lsspltxt="ls -p spl/*.txt"
$ $lsspltxt
spl/books.txt  spl/syllabus.txt
$ `$lsspltxt`
bash: spl/books.txt: Permission denied
$ cd spl/
$ $lsspltxt
ls: cannot access 'spl/*.txt': No such file or directory
$
```

Practice exercises

1. Investigate what the shell variables `PS1` and `PS2` do.
2. Type the following command at your bash prompt. What happens? Why does the directory not change?

```
‘cd /‘
```

3. Type the following commands. What do you learn?

```
x=‘ls -l‘
echo $x
echo "$x"
```

4. Investigate what the bash command `eval` does. Use the following commands as an example. Explain the outputs you get.

```
x="ls -l > lsout.txt"
$x
eval $x
$x
```

5. Investigate what the bash command `exec` does. Use the following commands as an example. Explain the outputs you get.

```
exec ls -l
exec ls -l &
exec ls -l | less
```

Practice exercises

6. Move to a directory containing at least one `.c` file. Enter the following commands. Explain the outputs you get.

```
‘ls -l *.c‘
‘ls -l "*.c"‘
```

7. Repeat the last exercise on the following commands.

```
‘ls -l *.c‘
echo $?
‘ls -l "*.c"‘
echo $?
```

8. Enter the following commands. Explain the outputs you get.

```
x=10
y="\$x"
echo $x
echo $y
echo "echo $y"
echo ’echo $y’
echo ‘echo $y‘
```

9. Repeat the last exercise with the following commands.

```
x=10
y="\$x"
echo eval $x
echo eval $y
eval echo $x
eval echo $y
```

Practice exercises

10. Define a function `f()` and call it as indicated. Explain the outputs you get.

```
function f () {
   echo $#
   echo $1
}
f *
f "*"
```

11. Repeat the last exercise with the function `g()`.

```
function g () {
   echo $#
   echo "$1"
}
g *
g "*"
```

12. Explain how you can read an array from a user in the following cases. Note that in neither of the cases, the user enters the array element by element. Note also that the number of elements in the array is not fixed a priori.

(a) The user enters the array in the format `(2 3 5 7 11 13 17 19)` in a single line.
(b) The user enters the array in the format `2 3 5 7 11 13 17 19` in a single line.
(c) The user enters the array in the format `2,3,5,7,11,13,17,19` in a single line.