

Lab2 - Shell Script

Brian Hsu @NASA2024

February 26, 2024

Outline

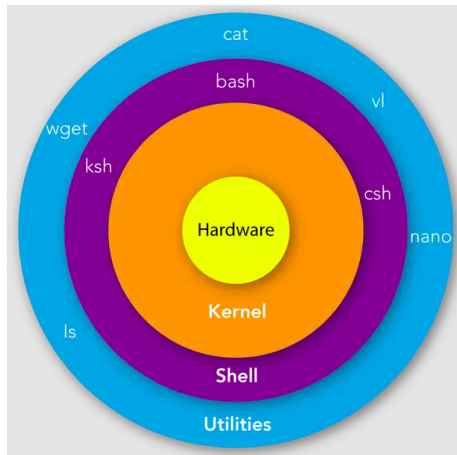
- 1 Introduction to Shell and Shell Script
- 2 Do Things Like A PRO
- 3 Shell Script - Syntax
- 4 Shell Script - Frequently Used Commands and Useful Tools
- 5 Exercise

Outline

- 1 Introduction to Shell and Shell Script
- 2 Do Things Like A PRO
- 3 Shell Script - Syntax
- 4 Shell Script - Frequently Used Commands and Useful Tools
- 5 Exercise

Shell

- Kernel: the core of OS
- Shell
 - the interface for user or program to access to the OS's services
 - often refer to the CLI (Command-Line Interface) shells
 - a sort of different shells: sh, bash, zsh, fish...



(Reference: <https://mindmajix.com/shell-scripting-tutorial>)

- Bash: Bourne-Again Shell
- Default login shell on most systems (ex: CSIE workstations)
- Will be used in this lab, HW, and semester
- Check your shell!

```
brianhsu:~$ echo $SHELL  
/bin/bash
```

- If you want to type a lot of commands, then directly typing it in shell is inconvenience.
- Type the commands in a file, and execute it.

Outline

- 1 Introduction to Shell and Shell Script
- 2 Do Things Like A PRO**
- 3 Shell Script - Syntax
- 4 Shell Script - Frequently Used Commands and Useful Tools
- 5 Exercise

Who to Ask

- Ask the `man` :
 - `man cmd` : show the manual page of the command `cmd`
 - Ex: `man cd` , `man ls` , ...
 - `man man` for more details
- Ask for help:
 - `cmd -h` , `cmd --help`
 - Ex: `cd --help` , `ls --help`
- Ask Google or ChatGPT: will be very helpful in this class
- [tldr](#) may help

- How to edit/execute codes on workstations?
 - VSCode (or other local editors) + scp ?
 - Learn Vim instead.
 - VIM IS THE BEST EDITOR IN THE WORLD!
- Vim tutorial
 - `vimtutor`
 - [Vim tutorial](#)
- Edit `~/.vimrc` to make Vim more convenient:
 - `map <F9> :w<bar>!gcc "%"`
 - `map <F10> :w<bar>!./a.out`



VSCode
+
SCP

Terminal
+
Vim



I Am Developer

@iamdeveloper



Following

I've been using Vim for about 2 years now, mostly because I can't figure out how to exit it.



- SSH: Secure SHell (secure for remote connection)
- Important:
 - Strong password
 - Give SSH key a try
- Keywords: `ssh-keygen`, `ssh-copy-id`, `ssh-agent`, `~/.ssh/config`, `keychain`
- Transferring file: `sftp`, `scp`

Outline

- 1 Introduction to Shell and Shell Script
- 2 Do Things Like A PRO
- 3 Shell Script - Syntax**
- 4 Shell Script - Frequently Used Commands and Useful Tools
- 5 Exercise

Hello World

- Open an editor and copy the following script:

```
#!/usr/bin/env bash
# The first line is shebang.
# Write comment after '#'.
echo 'Hello World!'
```

- Save it to a file (suppose it's `hello.sh`).

- Run it!

- First way: `bash hello.sh`
- Second way: `chmod +x hello.sh`
(remember to give it the execution permission)
`./hello.sh`

- Result:

```
brianhsu:~$ bash hello.sh
Hello World!
brianhsu:~$ chmod +x hello.sh
brianhsu:~$ ./hello.sh
Hello World!
```

- Shebang (=Hashbang): sharp + bang
- Specify the interpreter by adding shebang
- `#!/usr/bin/env bash` or `#!/bin/bash`
- Python is also an interpreter: `#!/usr/bin/env python`

Variables

- Assign value (no spaces in the syntax):

```
var=value
```

```
brianhsu:~$ a=b
brianhsu:~$ echo $a
b
brianhsu:~$ a = b
a: command not found
brianhsu:~$ a= b
b: command not found
brianhsu:~$ a =b
a: command not found
```

- Take value: `$var`, `${var}`
- echo with:
 - Single quote: without replacement
 - Double quote: with replacement

- I just assigned a variable, but `echo $variable` shows something else
- Try this yourself:

```
#!/usr/bin/env bash
var="echo Hello World!"
$var
echo var
echo $var
echo '$var'
echo "$var"
```

What is the output?

Variables

- Get the output of a command: `var=`cmd``, `var=$(cmd)`
- Get the exit code of the last command: `var=$?`
- Exit with a specific exit code: `exit code`
- Calculation
 - String concatenation: `ab`
 - Integer Arithmetic: `$((a+b))`, `((a*=b))`, `((++a))`, `((a-=1))`
 - String length: `${#a}`

Variables - Array

- Declare an array (IFS-sep):
`arr=("a" "b")`, `files=(`ls`)`
- Append to an array: `arr+=("c")`
- Assign: `arr[i]=value`
- Get the values of an array:
 - All items in `arr`: `${arr[@]}`
 - The size of `arr`: `${#arr[@]}`
 - The value of `arr[i]`: `${arr[i]}`
- Get the values of the arguments (can be seen as an array):
 - The argument of a script: `$1`, `$2`, ...
 - All arguments of a script: `$@`
 - The number of arguments: `$#`
- IFS (Internal Field Separator) (like `.split()` in python): `IFS=$' \t\n'` by default

Variables - Array

- Try the following code in a directory with more than 2 files.

```
#!/usr/bin/env bash
echo "$0 $1 $2"
echo "$(($1+$2))"
echo "$# $@"
arr=(`ls`)
echo ${arr[0]}
IFS=' '
arr=(`ls`)
echo ${arr[0]}
```

Save it to `script.sh` and run `./script.sh 12 34`.

Variables - Dictionary

- Declare with attributes: `declare [attribute-option] var=value`
- Delete a variable: `unset var`
- Declare a dictionary: `declare -A dict, dict=([abc]=def [123]='456')`
- Assign: `dict[key]=value`
- Get the values of an dictionary:
 - All items in `dict`: `${dict[@]}`
 - The size of `dict`: `${#dict[@]}`
 - The value of `dict[key]`: `${dict[key]}`
- What Are Bash Dictionaries on Linux, and How Do You Use Them?

Redirection

- Input into a variable: `read var`
- Redirection:
 - `<`: redirect STDIN to file
 - `>`: redirect STDOUT to file (cover)
 - `>>`: redirect STDOUT to file (append)
 - `2>`: redirect STDERR to file
 - `&>`: redirect STDOUT and STDERR to file
 - `>&`: redirect a stream to another

- What is the output and the content of each file after running the following code?

```
#!/usr/bin/env bash
echo 'Hello' > out
echo 'World' > out
echo 'Kitty' >> out
read a < out
echo $a
(read a && echo $a) < out > out1
cd -h > out2
cd -h 2> out3
(ls && cd -h) &> out4
cd -h 2>&1 > out5
cd -h > out6 2>&1
```

Redirection

- `cmd1 | cmd2`: pass the STDOUT of `cmd1` to the STDIN of `cmd2`

The following do almost the same thing:

- `cat messy.txt | sort | uniq > clean.txt`
- `cat messy.txt > tmp1; sort < tmp1 > tmp2`
`uniq < tmp2 > clean.txt; rm tmp1 tmp2`
- `cmd1; cmd2; ...`: use `;` to split the commands in a single line
- `cmd <<< str`: here-string, like redirecting STDIN to `str`
- `cmd &`: run `cmd` in background

If-Else

- If clause:

```
if cond; then
    cmd
fi
```

- If-else clause:

```
if cond1; then
    cmd1
elif cond2; then
    cmd2
else
    cmd3
fi
```

The spaces in the following syntaxes are important!

- Expression: `[[expr]]`

- Boolean operations:

- `[[! expr]]` : not
- `[[expr1]] && [[expr2]]` : and
- `[[expr1]] || [[expr2]]` : or

- Files:

- `[[-e FILE]]` : exists
- `[[-f FILE]]` : is file
- `[[-d FILE]]` : is directory
- `[[-L FILE]]` , `[[-h FILE]]` : is symbolic link

The spaces in the following syntaxes are important!

- Strings:

- `[[-z str]]` : is empty string
- `[[-n str]]` : is non-empty string
- `[[str1 == str2]]` : is equal
- `[[str1 != str2]]` : is unequal
- `[[str =~ regex]]` : `str` satisfies `regex`

- Numerical comparison:

- `[[a -eq b]]` : $a = b$
- `[[a -ne b]]` : $a \neq b$
- `[[a -lt b]]` : $a < b$
- `[[a -le b]]` : $a \leq b$
- `[[a -gt b]]` : $a > b$
- `[[a -ge b]]` : $a \geq b$

Loops

- `for i in range(10)`

```
for i in {0..9}; do
    echo $i
done
```

- `for(i=0; i<10; ++i)`

```
for ((i=0; i<10; ++i)); do
    echo $i
done
```

- `for i in arr`

```
for i in `ls`; do
    echo $i
done
```

- `while`

```
while cond; do
    cmd
done
```

- Use `continue` to continue, and use `break` to break.

Functions

- Declare:

```
func(){  
    cmd  
}
```

- Call: `func`

- Arguments:

- Passing: `func arg1 arg2 ...`
- Get the value: `$1`, `$2`, ... in the function declaration are the arguments

- Local variable: `local var`

- Example of `min` function:

```
#!/usr/bin/env bash  
min(){  
    local res=$1  
    for i in $@; do  
        if [[ $i -lt $res ]]; then  
            res=$i  
        fi  
    done  
    echo $res  
}  
echo `min 3 1 4 1 5 9`
```

Outline

- 1 Introduction to Shell and Shell Script
- 2 Do Things Like A PRO
- 3 Shell Script - Syntax
- 4 Shell Script - Frequently Used Commands and Useful Tools**
- 5 Exercise

Frequently Used Commands

- Get file attributes: `ls`, `stat`, `file`
- Working directory: `cd`, `pwd`
- Create files/directories: `touch`, `mkdir`
- Move/copy/delete: `mv`, `cp`, `rm`
- Permission: `chmod`, `chown`, `chgrp`, [ACL](#)
- Search: `find`, `which`, `whereis`, `locate`
- I/O: `echo`, `printf`, `cat`, `less`, `head`, `tail`, `read`
- String: `grep`, `awk`, `sed`, `cut`, `wc`, `diff`, `tr`
- Order: `sort`, `uniq`
- Process: `ps`, `top`, `kill`
- Runtime: `time`, `/usr/bin/time`, `timeout`, `ulimit`, `strace`, `sleep`
- Network utilities: `ifconfig`, `ip`, `ping`, `dig`, `traceroute`, `host`
- Connection: `nc`, `telnet`, `ssh`, `sftp`, `scp`
- Web: `wget`, `curl`
- Other: `bc`, `openssl`, `xargs`, `tee`, `trap`, [GlobPattern](#)

Regular Expression

- Regular expression (regex) is a kind of expression method describing some forms of texts.
- BRE (Basic Regular Expressions) is used as default by many commands (e.g. `sed`, `grep`).
- BRE syntax

Basic Regular Expressions

Element	Description
<code>c</code>	Match any character
<code>^</code>	Anchor the pattern to the beginning of a line
<code>\$</code>	Anchor the pattern to the end of a line
<code>.</code>	Match any single character
<code>[list]</code>	Match any single character in <code>list</code>
• <code>[^list]</code>	Match any single character not in <code>list</code>
<code>a*</code>	Match zero or more occurrences of <code>a</code>
<code>a{n}</code>	Match exactly <code>n</code> occurrences of <code>a</code>
<code>a{n,}</code>	Match at least <code>n</code> occurrences of <code>a</code>
<code>a{n,m}</code>	Match at least <code>n</code> and at most <code>m</code> occurrences of <code>a</code>
<code>r s</code>	Match either <code>r</code> or <code>s</code>
<code>(re)</code>	Grouping for precedence

Still A Lot to Learn

- Google
- Chatgpt
- Cheatsheet

Outline

- 1 Introduction to Shell and Shell Script
- 2 Do Things Like A PRO
- 3 Shell Script - Syntax
- 4 Shell Script - Frequently Used Commands and Useful Tools
- 5 Exercise**

Scenario - WA WA WA WA WA ...

224280181	Sep/21/2023 18:29 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang [#]	E - Hilbert's Hedge Maze	GNU C++17 (64)	Wrong answer on test 2	0 ms	0 KB
224277342	Sep/21/2023 17:59 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang [#]	H - Picking Up Steam	GNU C++17 (64)	Wrong answer on test 15	15 ms	0 KB
224276601	Sep/21/2023 17:51 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang [#]	E - Hilbert's Hedge Maze	GNU C++17 (64)	Wrong answer on test 2	0 ms	0 KB
224276301	Sep/21/2023 17:48 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang [#]	E - Hilbert's Hedge Maze	GNU C++17 (64)	Wrong answer on test 2	0 ms	0 KB
224275030	Sep/21/2023 17:34 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang [#]	E - Hilbert's Hedge Maze	GNU C++17 (64)	Wrong answer on test 2	0 ms	0 KB
224274214	Sep/21/2023 17:26 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang [#]	H - Picking Up Steam	GNU C++17 (64)	Wrong answer on test 13	0 ms	0 KB
224274072	Sep/21/2023 17:24 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang [#]	E - Hilbert's Hedge Maze	GNU C++17 (64)	Wrong answer on test 2	0 ms	0 KB
224272774	Sep/21/2023 17:12 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang [#]	H - Picking Up Steam	GNU C++17 (64)	Wrong answer on test 13	0 ms	0 KB

Scenario - WA WA WA WA WA ...

224272623	Sep/21/2023 17:11 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang #	H - Picking Up Steam	GNU C++17 (64)	Wrong answer on test 13	0 ms	0 KB
224271140	Sep/21/2023 16:57 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang #	H - Picking Up Steam	GNU C++17 (64)	Wrong answer on test 9	0 ms	0 KB
224268953	Sep/21/2023 16:38 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang #	H - Picking Up Steam	GNU C++17 (64)	Wrong answer on test 13	0 ms	0 KB
224267197	Sep/21/2023 16:20 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang #	H - Picking Up Steam	GNU C++17 (64)	Wrong answer on test 3	0 ms	0 KB
224266697	Sep/21/2023 16:15 ^{UTC+8}	too soft: Brian_Hsu, 2qbingxuan, casperwang #	H - Picking Up Steam	GNU C++17 (64)	Wrong answer on test 3	0 ms	0 KB

- The code with correct time complexity is complicated.
- However, it is easy to write the code using brute force.

How to Debug

- The WA test case might help.
- The judge does not show the WA test case.
- Generate it yourself.
- Compare with the output of your brute force code.

Generator

- Randomly generate some tests in the given range.
- Example generator `gen.c` of [Print Two Numbers](#).

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    srand(atoi(argv[1]));
    printf("%d\n%d\n", rand(), rand());
    return 0;
}
```

- Compile `gen.c` to `gen`, and run `./gen arg`.
- The output is determined by the argument `arg`.

Brute Force Solution

- A code that generates the correct output.
- Example brute force solution `sol.c` of [Print Two Numbers](#).

```
#include <stdio.h>

int main(){
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d\n%d\n", b, a);
    return 0;
}
```

The WA Code

- The code that got WA on the judge.
- Example WA code `wa.c` of [Print Two Numbers](#).

```
#include <stdio.h>

int main(){
    int a, b;
    scanf("%d%d", &a, &b);
    if(a+b&1)printf("%d\n%d\n", b, a);
    else printf("%d\n%d\n", a, b);
    return 0;
}
```

- Write a shell script (called `debugger.sh`).
- Usage: `./debugger.sh <Generator> <Code 1> <Code 2> <time>`
- The shell script should do the following things:
 - Compile `<Generator>`, `<Code 1>`, `<Code 2>` into `gen`, `a`, `b`, respectively.
 - For `i=1` to `<time>`, use the output generated by `./gen i` as the input `in.txt` of `a` and `b`.
 - If the outputs of running `a` and `b` are different, then output the debug message (the format is in the next page), and then `debugger.sh` should exit.

- Output format:

- First line: `Test $i:`
- Second line: `Input:`
- Print the content of `in.txt` with 20 '-' each above and below.
- Next line: `Output of <Code 1>:`
- Print the output of running `a` with 20 '-' each above and below.
- Next line: `Output of <Code 2>:`
- Print the output of running `b` with 20 '-' each above and below.

Debugger

- Example output of `./debugger.sh gen.c sol.c wa.c 10`:

```
Test 4:
```

```
Input:
```

```
-----
```

```
287724083
```

```
1968078301
```

```
-----
```

```
Output of sol.c:
```

```
-----
```

```
1968078301
```

```
287724083
```

```
-----
```

```
Output of wa.c:
```

```
-----
```

```
287724083
```

```
1968078301
```

```
-----
```

Time Limit

- Time limit: 1 minute.
- $\text{<time>} \leq 1000$.
- If <Code A> , <Code B> are guaranteed to finish running in s seconds, then $s \times \text{<time>} \leq 50$ is guaranteed.

- Deadline: 2024/2/29 23:59
- Submit your code [here](#).
- Check your result [here](#).
- Use base64 to encode a file: `base64 FILE`
- Calculate the sha1sum of a file: `sha1sum FILE`
- 0 point for plagiarism.
- 0 point for any malicious behavior in your script.
- Your script should NOT create nor remove things that are not in the current working directory.
- Your script will be run on the workstation.

- [02/27 11:04]: Notice the output of the following code:

```
#!/usr/bin/env bash
func(){
    echo 'Hello World!'
    echo
    echo
}
func
echo "`func`"
```

Acknowledgement

- Hsin-Mu
- HY, Lin (2023 TA)
- Brian Tsai (2022 TA)
- Giver (2021 TA)
- Wu-Jun Pei (2020 TA)
- Tsung-Han Wu (2019 TA)
- Kai-Ling Lo (2018 TA)
- The entire NASA team