

# Shell Programming

---

Put distinctive simple tools together to accomplish  
your goal...

zswu

# Outline

- 
- ❑ Variable pre-operations
  - ❑ args, argc in Shell Scripts
  - ❑ Arithmetic and Logics
    - Test commands
  - ❑ Control Structures: if-else, switch-case, for/while loops
  - ❑ Input/output: Read from screen
  - ❑ Defining Functions & Parsing Arguments
  - ❑ Error Handling and Debug tool (sh -x)
  - ❑ A Shell Script Sample: Failure Detection on Servers
  
  - ❑ Appendix: Regular Expression
  - ❑ Appendix B: sed and awk

# Bourne Shell

---

□ We use Bourne Shell in this slide.

- % echo \$SHELL
- /usr/local/bin/bash
  
- % sh
- \$

# Executable script

---

## □ Shebang

- `#!/bin/sh`
- `#!/bin/bash`
- `#!/usr/local/bin/bash`
- `#!/usr/bin/env bash`

## □ Execution

- `$ sh test.sh`
- `$ chmod a+x test.sh`
- `$ ./test.sh`

# Shell variables (1)

## □ Assignment

	Syntex	Scope
Variable	<code>my=test</code>	Process
Local variable	<code>local my=test</code>	Function
Environment variable	<code>export my</code>	Process and subprocess

- Example:
  - `$ export PAGER=/usr/bin/less`
  - `$ current_month=`date +%m``
  - `$ myFun() { local arg1="$1" }`

# Shell variables (2)

There are two ways to call variable...

“\${var}”... why?

## □ Access

- % echo “\$PAGER”
- % echo “\${PAGER}”
- Use {} to avoid ambiguity
  - % temp\_name=“haha”
  - % temp=“hehe”
  - % echo \$temp
    - hehe
  - % echo \$temp\_name
    - haha
  - % echo \${temp}\_name
    - hehe\_name
  - % echo \${temp\_name}
    - haha

No space beside ‘=’

More clear...

# Quotation marks

---

## □ Single quotes('xxx')

- Preserves the literal value of each character within the quotes
- \$ echo 'echo \$USER'  
  > echo \$USER

## □ Double quotes("xxx")

- Parse special character, like: \$ ` \
- \$ echo "echo \$USER"  
  > echo zswu

## □ Grave Accent(`xxx`)

- The stdout of the command
- \$ echo `echo \$USER`  
  > zswu

# Shell variable operator (1)

※ Bad == not Good

“Bad” : var is not set or the value is null

“Good” : var is set and is not null

operator	Description
<code> \${var:=value}</code>	If “Bad”, use the value and assign to var
<code> \${var:+value}</code>	If “Good”, use the value instead else <u>null value is used but not assign to var</u>
<code> \${var:-value}</code>	If “Good”, use the value of var else use the value <u>but not assign to var</u>
<code> \${var:?value}</code>	If “Bad”, <b>print value</b> and <u>shell exits</u>

Print → stderr

The command stops  
immediately

“Parameter Expansion” in sh(1)

# Shell variable operator (2)

❑ Ex:

```
#!/bin/sh
```

```
var1="haha"  
echo "01" ${var1:+hehe}"  
echo "02" ${var1}  
echo "03" ${var2:+hehe}"  
echo "04" ${var2}  
echo "05" ${var1:="hehehe"}  
echo "06" ${var1}  
echo "07" ${var2:="hehehe"}  
echo "08" ${var2}  
echo "09" ${var1:-"he"}  
echo "10" ${var1}  
echo "11" ${var3:-"he"}  
echo "12" ${var3}  
echo "13" ${var1:??"hoho"}  
echo "14" ${var1}  
echo "15" ${var3:??"hoho"}  
echo "16" ${var3}
```

❑ Result:

01 hehe
02 haha
03
04
05 haha
06 haha
07 hehehe
08 hehehe
09 haha
10 haha
11 he
12
13 haha
14 haha
hoho
16

# Shell variable operator (3)

operator	description
<code> \${#var}</code>	String <u>length</u>
<code> \${var#pattern}</code>	Remove the <u>smallest prefix</u>
<code> \${var##pattern}</code>	Remove the <u>largest prefix</u>
<code> \${var%pattern}</code>	Remove the <u>smallest suffix</u>
<code> \${var%%pattern}</code>	Remove the <u>largest suffix</u>

```
#!/bin/sh
```

These operators do not change the value of var...

```
var="Nothing happened end closing end"
```

```
echo ${#var}
echo ${var#*ing}
echo ${var##*ing}
echo ${var%end*}
echo ${var%%end*}
```

Results:

32	
happened end closing end	
end	
Nothing happened end closing	
Nothing happened	

# Predefined shell variables

Similar to C program's "Int main(argc, args)" – **arguments of program**,  
e.g. ls -a ~

- Environment Variables
- Other useful variables:

sh	description
\$#	<u>Number</u> of positional arguments
\$0	Command name (Ex: What command user exec your script)
\$1, \$2, ..	Positional <u>arguments</u>
\$* / \$@	<u>List of positional arguments</u> (useful in for loop) \${*:2} : Get the list of argument after \$2
\$?	Return code from <u>last command</u>
\$\$	Process number of <u>current command (pid)</u>
\$!	Process number of <u>last background command</u>

# Usage of \$\* and \$@

- The difference between \$\* and \$@
  - \$\* : all arguments are formed into a long string
  - \$@ : all arguments are formed into separated strings
- Examples: test.sh

```
for i in "$*" ; do  
    echo $i  
done
```

```
% test.sh 1 2 3
```

1 2 3

```
for i in "$@" ; do  
    echo $i  
done
```

```
% test.sh 1 2 3
```

1  
2  
3

# test command

Checking things for us... e.g. file status, statements

---

## □ test(1)

- test expression
- [ expression ]
- Test for: file, string, number

## □ Test and return 0 (true) or 1 (false) in \$?

- % test -e News ; echo \$?                      → \$? To obtain the return code
  - If there exist the file named “News”
- % test "haha" = "hehe" ; echo \$?
  - Whether “haha” **equal** “hehe”
- % test 10 -eq 11 ; echo \$?
  - Whether 10 **equal** 11

# Details on the capability of test command – File test

- ❑ -e file
    - True if file exists (regardless of type)
  - ❑ -s file
    - True if file exists and has a size greater than zero
  - ❑ -b file
    - True if file exists and is a block special file
  - ❑ -c file
    - True if file exists and is a character special file
  - ❑ -d file
    - True if file exists and is a directory
  - ❑ -f file
    - True if file exists and is a regular file
  - ❑ -p file
    - True if file is a named pipe (FIFO)
  - ❑ -L file
    - True if file exists and is a symbolic link
  - ❑ -S file
    - True if file exists and is a socket
  - ❑ -r file
    - True if file exists and is readable
  - ❑ -w file
    - True if file exists and is writable
  - ❑ -x file
    - True if file exists and is executable
  - ❑ -u file
    - True if file exists and its set user ID flag is set
  - ❑ -g file
    - True if file exists and its set group ID flag is set
  - ❑ -k file
    - True if file exists and its sticky bit is set
  - ❑ -O file
    - True if file exists and its owner matches the effective user id of this process
  - ❑ -G file
    - True if file exists and its group matches the effective group id of this process
  - ❑ file1 -nt file2
    - True if file1 exists and is newer than file2
  - ❑ file1 -ot file2
    - True if file1 exists and is older than file2
  - ❑ file1 -ef file2
    - True if file1 and file2 exist and refer to the same file
- Hard links to same file..

# Details on the capability of test command – String test

- ❑ -z string
  - True if the length of string is zero
- ❑ -n string
  - True if the length of string is nonzero
- ❑ string
  - True if string is not the null string
- ❑ s1 = s2 (though some implementation recognize ==)
  - True if the strings s1 and s2 are identical
- ❑ s1 != s2
  - True if the strings s1 and s2 are not identical
- ❑ s1 < s2
  - True if string s1 comes before s2 based on the binary value of their characters
- ❑ s1 > s2
  - True if string s1 comes after s2 based on the binary value of their characters

## Details on the capability of test command – Number test

- $n1 \text{-eq } n2$        $==, !=, >, <, >=, <=$  fashion does not apply here...
  - True if the integers  $n1$  and  $n2$  are algebraically **equal**
- $n1 \text{-ne } n2$ 
  - True if the integers  $n1$  and  $n2$  are **not** algebraically **equal**
- $n1 \text{-gt } n2$ 
  - True if the integer  $n1$  is algebraically **greater than** the integer  $n2$
- $n1 \text{-ge } n2$ 
  - True if the integer  $n1$  is algebraically **greater than or equal to** the integer  $n2$
- $n1 \text{-lt } n2$ 
  - True if the integer  $n1$  is algebraically **less than** the integer  $n2$
- $n1 \text{-le } n2$ 
  - True if the integer  $n1$  is algebraically **less than or equal to** the integer  $n2$

# test command – combination

---

## □ ! expression

- True if expression is false.
- $\$[ ! A == B ] \Rightarrow$  Test expression
- $\$![ A == B ] \Rightarrow$  Invert test command result

## □ expression1 -a expression2

- True if both expression1 and expression2 are true.
- $\$[ A == B -a C == D ]$
- $\$[ A == B ] \&& [ C == D ]$

## □ expression1 -o expression2

- True if either expression1 or expression2 are true.
- The **-a operator has higher precedence than the -o operator.**
- $\$[ A == B -o C == D ]$
- $\$[ A == B ] \| [ C == D ]$

## test command – in script

- Add space beside = <= != [ ]...
  - \$ [A=B] # error
  - \$ [ A=B ] # error
  - \$ [A = B] # error
- If the var may be null or may not be set, add “”
  - \$ [ \$var = “A” ] may be parsed to [ = “A” ] and cause **syntax error!!**
  - \$ [ “\$var” = “A” ] become [ “” = “A” ]

```
if [ “$var” = “hehe” ] ; then
    echo ‘$var equals hehe’
else
    echo ‘$var doesn’t equal hehe’
fi
```

# expr command (1)

---

# AND - OR – NOT

```
$ [ 1 -eq 2 ] || [ 1 -eq 1 ] ; echo $?  
0
```

```
$ [ 1 -eq 2 ] && [ 1 -eq 1 ] ; echo $?  
1
```

```
$ [ 1 -eq 1 ] || [ 1 -eq 2 ] ; echo $?  
0
```

```
$ ! [ 1 -eq 2 ] ; echo $?  
0
```

```
$ [ 1 -eq 1 ] && [ 1 -eq 2 ] ; echo $?  
1
```

```
$ [ 1 -eq 2 ] ; echo $?  
1
```

## expr command (2)

---

- \$ expr1 && expr2 , if expr1 is false then expr2 won't be evaluate
- \$ expr1 || expr2 , if expr1 is true then expr2 won't be evaluate
- Ex:
  - \$ [ -e SomeFile ] && rm SomeFile
  - \$ checkSomething || exit 1

# Arithmetic Expansion

---

```
echo $(( 1 + 2 ))          3
```

```
a=8                      // a=8
```

```
a=$(( $a + 9 ))        // a=17
```

```
a=$(( $a + 17 ))       // a=34
```

```
a=$(( $a + 9453 ))     // a=9487
```

```
echo $a                  9487
```

# if-then-else structure



```
if [ test conditions ] ; then  
    command-list  
elif [ test contitions ] ; then  
    command-list  
else  
    command-list  
fi  
# Or in one line  
if [ a = a ]; then echo "Yes"; else echo "No"; fi
```

# switch-case structure (1)

```
case $var in
    value1)
        action1
    ;;
    value2)
        action2
    ;;
    value3|value4)
        action3
    ;;
    *)
        default-action
    ;;
esac
```

```
case $sshd_enable in
    [Yy][Ee][Ss])
        action1
    ;;
    [Nn][Oo])
        action2
    ;;
    *)
        ???
    ;;
esac
```

# For loop

sh



```
for var in var1 var2 ...; do  
    action  
done
```

```
a=""  
for var in `ls`; do  
    a="$a $var"  
done  
echo $a
```

```
for i in A B C D E F G; do  
    mkdir $i;  
done
```

# While loop

---

sh

while [...] ; do

    action

done

break

continue

while read name ; do

    echo "Hi \$name"

done

# Read from stdin

---

```
#!/bin/sh
echo -n "Do you want to 'rm -rf /' (yes/no)? "
read answer
case $answer in
[Yy][Ee][Ss])
    echo "Hahaha"
;;
[Nn][Oo])
    echo "No~~~"
;;
*)
    echo "removing..."
;;
esac
```

## Create tmp file/dir

---

- `TMPDIR=`mktemp -d tmp.XXXXXX``
- `TMPFILE=`mktemp ${TMPDIR}/tmp.XXXXXX``
- `echo "program output" >> ${TMPFILE}`

# functions (1)

---

- Define function

```
function_name () {  
    command_list  
}
```

- Removing function definition

```
unset function_name
```

- Function execution

```
function_name
```

- Function definition is local to the current shell

※ Define the function before first use...

# functions (2) - scoping

---

```
func () {  
    # global variable  
    echo $a  
    a="hello"  
}  
a="5566"
```

```
func  
echo $a
```

Result:  
5566  
hello

```
func () {  
    # global variable  
    local a="hello"  
    echo $a  
}  
a="5566"
```

```
func  
echo $a
```

Result:  
hello  
5566

## functions (3) - arguments check

---

```
func () {  
    if [ $# -eq 2 ] ; then  
        echo $1 $2  
    else  
        echo "Wrong"  
    fi  
}  
func  
func hi  
func hello world
```

Result:

Wrong

Wrong

hello world

# functions (4) - return value

---

```
func () {  
    if [ $# -eq 2 ] ; then  
        return 0  
    else  
        return 2  
    fi  
}  
func  
echo $?  
func hello world  
echo $?
```

Result:

2  
0

# Scope

---

- ❑ Local var can only be read and written inside the function.
- ❑ Subprocess can only read the environment variable, the modification of the variable will **NOT** be effective to the current process. (Subprocess may include some PIPE execution)
- ❑ If something wrong, try to print every variable.
- ❑ Ex:
  - \$ a=10
  - \$ export b=20
  - \$ cat SomeFile | while read line; do
  - >           echo “\$a \$b \$line” # a is 10, b is 20
  - >           b=\$((b+1))
  - > done
  - > echo \$b # b is still 20

# Parsing arguments

## □ Use getopt (recommended)

```
#!/bin/sh

while getopts abcf: op ; do
    echo "${OPTARG}-th arg"

    case $op in
        a|b|c)
            echo "one of ABC" ;;
        f)
            echo $OPTARG ;;
        *)
            echo "Default" ;;
    esac
done
```

```
$ ./test.sh -a -b -c -f gg
2-th arg
one of ABC
3-th arg
one of ABC
4-th arg
one of ABC
6-th arg
gg
```

“::” means additional arg.  
\$OPTARG: content of arguments  
\$OPTIND: index of arguments

# Handling Error Conditions

---

## □ Internal error      ← program crash

- Caused by some command's failing to perform
  - User-error
    - Invalid input
    - Unmatched shell-script usage
  - Command failure

## □ External error      ← signal from OS

- By the system telling you that some system-level event has occurred by sending signal

# Handling Error Conditions – Internal Error

□ Ex:

```
#!/bin/sh
UsageString="Usage: $0 -man=val1 -woman=val2"

if [ $# != 2 ] ; then
    echo "$UsageString"
else
    echo "ok!"
    man=`echo $1 | cut -c 6-` ← start from char6
    woman=`echo $2 | cut -c 8-`
    echo "Man is ${man}"
    echo "Woman is ${woman}"
fi
```

program name

How about  
c but not -c?

→ Handling the errors yourself...

# Handling Error Conditions – External Error (1)



## □ Using trap in Bourne shell

- trap [command-list] [signal-list]
  - Perform command-list when receiving any signal in signal-list

Usag: trap “[commands]” list of signals looking for...

```
trap “rm tmp*; exit0” 1 2 3 14 15
```

```
trap "" 1 2 3    Ignore signal 1 2 3
```

# Handling Error Conditions – External Error (2)

#	Name	Description	Default	Catch	Block	Dump core
1	SIGHUP	Hangup	Terminate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
2	SIGINT	Interrupt (^C)	Terminate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
3	SIGQUIT	Quit	Terminate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
9	SIGKILL	Kill	Terminate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10	SIGBUS	Bus error	Terminate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
11	SIGSEGV	Segmentation fault	Terminate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
15	SIGTERM	Soft. termination	Terminate	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
17	SIGSTOP	Stop	Stop	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
18	SIGTSTP	Stop from tty (^Z)	Stop	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
19	SIGCONT	Continue after stop	Ignore	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

# Debugging Shell Script

Debug tools in sh...

❑ Ex:

```
#!/bin/sh -x
```

```
var1="haha"
echo "01" ${var1:+hehe"}
echo "02" ${var1}
echo "03" ${var2:+hehe"}
echo "04" ${var2}
echo "05" ${var1:="hehehe"}
echo "06" ${var1}
echo "07" ${var2:="hehehe"}
echo "08" ${var2}
echo "09" ${var1:-"he"}
echo "10" ${var1}
echo "11" ${var3:-"he"}
echo "12" ${var3}
echo "13" ${var1:?hoho"}
echo "14" ${var1}
echo "15" ${var3:?hoho"}
echo "16" ${var3}
```

Debug mode



❑ Result:

```
+ var1=haha
+ echo 01 hehe
01 hehe
+ echo 02 haha
02 haha
+ echo 03
03
+ echo 04
04
+ echo 05 haha
05 haha
+ echo 06 haha
06 haha
+ echo 07 hehehe
07 hehehe
+ echo 08 hehehe
08 hehehe
+ echo 09 haha
09 haha
+ echo 10 haha
10 haha
+ echo 11 he
11 he
+ echo 12
12
+ echo 13 haha
13 haha
+ echo 14 haha
14 haha
hoho
```

Debug msgs.

print out the

**substitution results...**

# Useful tools

---

- ❑ head
- ❑ tail
- ❑ grep
- ❑ find
- ❑ ps
- ❑ xargs

# Shell Script Examples

---

# check alive (1)

---

## □ ping

- /sbin/ping -c 3 bsd1.cs.nctu.edu.tw

```
PING bsd1.cs.nctu.edu.tw (140.113.235.131): 56 data bytes
```

```
64 bytes from 140.113.235.131: icmp_seq=0 ttl=60 time=0.472 ms
```

```
64 bytes from 140.113.235.131: icmp_seq=1 ttl=60 time=0.473 ms
```

```
64 bytes from 140.113.235.131: icmp_seq=2 ttl=60 time=0.361 ms
```

```
--- bsd1.cs.nctu.edu.tw ping statistics ---
```

```
3 packets transmitted, 3 packets received, 0% packet loss
```

```
round-trip min/avg/max/stddev = 0.361/0.435/0.473/0.053 ms
```

# check alive (2)

```
#!/bin/sh  
# [Usage] isAlive.sh ccbsd1
```

```
Usage="[Usage] $0 host"  
temp="$1.ping"  
Admin="liuyh"  
count="20"
```

```
if [ $# != 1 ] ; then  
    echo $Usage  
else  
    /sbin/ping -c ${count:=10} $1 | /usr/bin/grep 'transmitted' > $temp  
    Lost=`awk -F" " '{print $7}' $temp | awk -F"%" '{print $1}'`
```

```
if [ ${Lost:=0} -ge 50 ] ; then  
    mail -s "$1 failed" $Admin < $temp  
fi  
/bin/rm $temp  
fi
```

default 10 times

Grep "tran..."

wrtie to the temp file

- awk on \$temp using space as delimiter
- How many % packet loss?

Mail and del. \$temp

# Appendix A: Regular Expression

---

pattern matching

# Regular Expression (1)

---

## □ Informal definition

- Basis:
  - A single character “a” is a R.E.
- Hypothesis
  - If r and s are R.E.
- Inductive
  - Union:  $r + s$  is R.E
    - Ex:  $a + b$
  - Concatenation:  $rs$  is R.E.
    - Ex:  $ab$
  - Kleene closure:  $r^*$  is R.E.
    - Ex:  $a^*$

# Regular Expression (2)

## □ Pattern-matching

- Special operators

operator	Description
.	Any single character
[]	Any character in []
[^]	Any character <b>not</b> in []
^	<u>start</u> of a line
\$	<b>end</b> of a line
*	zero or more
?	zero or <b>one</b>
+	<b>one</b> or more
{m,n}	At least <b>m</b> times and at most <b>n</b> times
{m,}	At least <b>m</b> times.
{m}	<b>Exactly</b> m times.
\	Escape character

# Regular Expression (3)

## □ Examples

- r.n
  - Any 3-character string that start with r and end with n
    - r1n, rxn, r&n will match
    - r1xn, axn will not match
- ..Z..
  - Any 5-character strings that have Z as 3<sup>rd</sup> character
    - aeZoo, 12Zos will match
    - aeooZ, aeZoom will not match
- r[a-z]n
  - Any 3-character string that start with r and end with n and the 2<sup>nd</sup> character is an alphabet
    - rxn will match
    - r1n, r&n will not match

# Regular Expression (4)

---

## □ Examples

- $^{\wedge}\text{John}$ 
  - Any string starts with John
    - John Snow -> will match
    - Hi John -> will not match
- $[\text{En}][\text{Nn}][\text{Dd}]\$$ 
  - Any string ends with any combination of “end”
- $[\text{A-Za-z0-9}]^+$ 
  - String of characters

# Regular Expression (5)

---

## □ Utilities using RE

- grep
- awk
- sed
- find

## □ Different tools, different RE

- BRE (Basic)
- ERE (Extended)
- PCRE (Perl Compatible)
- [https://en.wikipedia.org/wiki/Regular\\_expression#Standards](https://en.wikipedia.org/wiki/Regular_expression#Standards)

## Appendix B: sed and awk

---

Details on using sed and awk...

# sed – Stream EDitor (1)

---

## □ sed(1)

- sed –e “command” –e “command”... file
- sed –f script-file file
  - Sed will (1) read the file line by line and (2) do the commands, then (3) output to stdout
  - e.g. sed -e '1,10d' -e 's/yellow/black/g' yel.dat

## □ Command format

- [address1[,address2]]function[argument]
  - From address 1 to address 2
  - Do what action

## □ Address format

- n ➔ line number
- /R.E./ ➔ the line that matches R.E

# sed – Stream EDitor (2)

---

- Example of address format
  - sed -e 10d
  - sed -e /man/d
  - sed -e 10,100d
  - sed -e 10,/man/d
    - Delete line from line 10 to the line contain “man”

# sed – Stream EDitor

## Function: substitution (1)

---

### □ substitution

- Syntax
  - s/pattern/replace/flags
- Flags
  - N: Make the substitution only for the N'th occurrence
  - g: replace all matches
  - p: print the matched and replaced line
  - w: write the matched and replaced line to a file

# sed – Stream EDitor

## Function: substitution (2)

### □ Ex:

- sed –e ‘s/liuyh/LIUYH/2’ file
- sed –e ‘s/liuyh/LIUYH/g’ file
- sed –e ‘s/liuyh/LIUYH/p’ file
- sed –n –e ‘s/liuyh/LIUYH/p’ file
- sed –e ‘s/liuyh/LIUYH/w wfile’ file

file

I am jon

I am john

I am liuyh

I am liuyh

I am nothing

# sed – Stream EDitor

## Function: delete

---

### □ delete

- Syntax:  
[address]d

### □ Ex:

- sed -e 10d
- sed -e /man/d
- sed -e 10,100d
- sed -e 10,/man/d

## sed – Stream EDitor

Function: **append, insert, change**

### □ append, insert, change

- Syntax:

[address]a\  
text

[address]i\  
text

- insert → insert before the line
- change → replace whole line

### □ Ex:

- sed -f sed.src file

sed.src

/liuyh/i \  
Meet liuyh, Hello

file  
I am jon  
I am john  
I am liuyh  
I am liuyh  
I am nothing

Results:  
I am jon  
I am john  
Meet liuyh, Hello  
**I am liuyh**  
Meet liuyh, Hello  
**I am liuyh**  
I am nothing

# sed – Stream EDitor

## Function: print

---

### □ print

- Syntax:

[addr1, addr2]p

### □ Ex:

- `sed -n -e '/^liuyh/p'` Print out the lines that begins with liuyh

**-n:** By default, each line of input is echoed to the standard output after all of the commands have been applied to it. The `-n` option suppresses this behavior.

# awk

## □ awk(1)

- awk [-F fs] [ ‘awk\_program’ | -f program\_file] [data\_file .....]
  - awk will read the file line by line and evaluate the pattern, then do the action if the test is true
  - Ex:
    - awk ‘{print “Hello World”}’ file
    - awk ‘{print \$1}’ file

## □ Program structure

- pattern { action }
- missing pattern means always matches
- missing { action } means print the line

Amy	32	0800995995	nctu.csie	
\$1	\$2	\$3		\$4

# awk – Pattern formats

---

## □ pattern formats

- Regular expression
  - awk '/[0-9]+/ {print "This is an integer"}'
  - awk '/[A-Za-z]+/ {print "This is a string"}'
  - awk '/^\$/ {print "this is a blank line."}'
- BEGIN
  - before reading any data
    - awk 'BEGIN {print "Nice to meet you"}'
- END
  - after the last line is read
    - awk 'END {print "Bye Bye"}'

# awk – action format

## □ Actions

- Print
- Assignment
- if( expression ) statement [; else statement2]
  - awk ' { if( \$2 ~ /am/ ) print \$1 }' file
- while( expression ) statement
  - awk 'BEGIN {count=0} /liuyh/ {while (count < 3) {print count;count++}}' file
  - awk 'BEGIN {count=0} /liuyh/ {while (count < 3) {print count;count++;count=0}}' file

var usage: no need for “\$”  
reset count after printing
- for ( init ; test ; incr ) action
  - awk '{for (i=0;i<3;i++) print i}' file

# awk – built-in variables (1)

---

- \$0, \$1, \$2, ...
  - Column variables
- NF
  - Number of fields in current line
- NR
  - Number of line processed
- FILENAME
  - the name of the file being processed
- FS
  - Field separator, set by **-F**
- OFS
  - Output field separator

# awk – built-in variables (2)

---

## □ Ex:

- awk 'BEGIN {FS=":"} /liuyh/ {print \$3}' /etc/passwd  
➤ 1002
- awk 'BEGIN {FS=":"} /^liuyh/{print \$3 \$6}' /etc/passwd  
➤ 1002/home/liuyh
- awk 'BEGIN {FS=":"} /^liuyh/{print \$3 " " \$6}' /etc/passwd  
➤ 1002 /home/liuyh
- awk 'BEGIN {FS=":" ;OFS="==" } /^liuyh/{print \$3 ,\$6}' /etc/passwd  
➤ 1002==/home/liuyh

# Reference

---

- awk(1)
- sed(1)
- <http://www.grymoire.com/Unix/Awk.html>
- <http://www.grymoire.com/Unix/Sed.html>
- [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)