

SHELL SCRIPTING

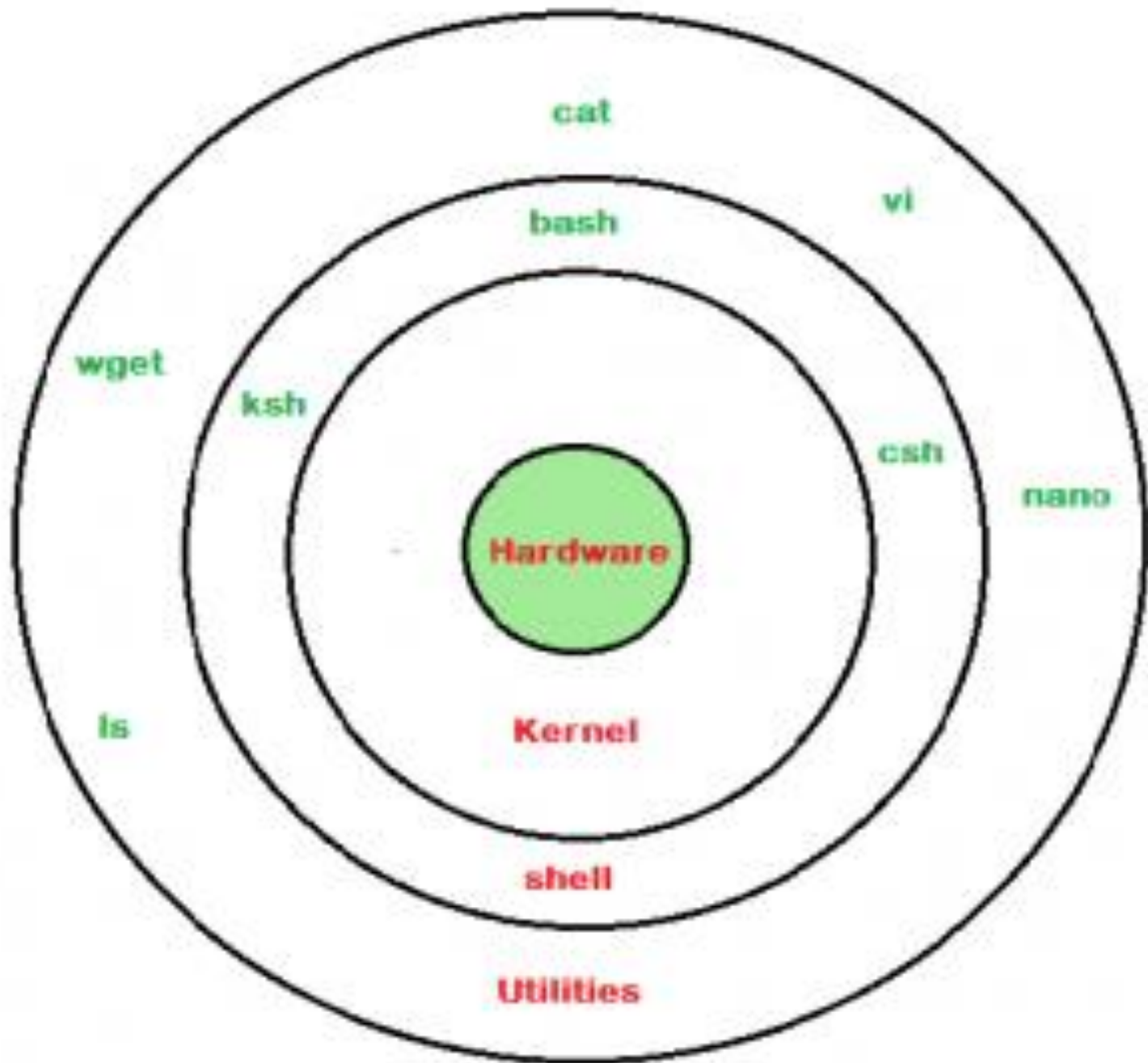


What is Shell

- A shell is special user program which provide an interface to user to use operating system services.
- Shell accept human readable commands from user and convert them into something which kernel can understand.
- It is a command language interpreter that execute commands read from input devices such as keyboards or from files.

What is Shell

- A shell is special user program which provide an interface to user to use operating system services.
- Shell accept human readable commands from user and convert them into something which kernel can understand.
- It is a command language interpreter that execute commands read from input devices such as keyboards or from files.



TYPES OF SHELLS

There are four shells

- Bourne shell(sh),
- Korn shell(ksh),
- C shell(csh) and
- Bourne Again Shell (bash).

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

VARIABLE

- A variable is a character string to which we assign a value.
- The value assigned could be a number, text, filename, device, or any other type of data
- Valid variables
 - `_abc`
 - `Ab_c`
 - `Ab_1`
- Invalid Variables
 - `1_ab`
 - `-ab`
 - `Ab-cd`
 - `Ab_c!`

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

OPERATORS

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

ARITHMETIC OPERATORS

- shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **expr** or **bc**.
- `C=`expr 1 + 1``
- There must be spaces between operators and expressions. For example, `2+2` is not correct; it should be written as `2 + 2`.
- The complete expression should be enclosed between `` ``, called the backtick.

FLOATING POINT

- Bc - **bc** command is used for command line calculator.
- Input : **\$ echo "12+5" | bc**
- Output : **17**
- Input : **\$ echo "10/2" | bc**
- Output : **5**
- Input: **\$ echo "5.1*2.1" | bc**
- Output : **10.7**
- Input: **\$ echo "scale=1;5/2" | bc**
- Output : **10.7**
- **[root@localhost ~]# div=`echo "scale=1;5/2" | bc`**
- **[root@localhost ~]# echo \$div**
- **[root@localhost ~]# 2.5**

- **How to store the result of complete operation in variable?**

Example:

- **Input:**
- **\$ x=`echo "12+5" | bc`**
- **\$ echo \$x**
- **Output:17**

USIING EXPR

```
#!/bin/bash
```

```
read -p "enter two numbers= " a b
```

```
add=`expr $a + $b`
```

```
echo "add=$add"
```

```
dev=`expr $a / $b`
```

```
echo "dev=$dev"
```

USING BC

```
#!/bin/bash
```

```
read -p "enter two numbers= " a b
```

```
add=`echo "$a+$b" | bc`
```

```
echo "add=$add"
```

```
dev=`echo "scale=1;$a/$b" | bc`
```

```
echo "dev=$dev"
```


ARITHMETIC OPERATORS

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	<code>`expr \$a + \$b`</code>
- (Subtraction)	Subtracts right hand operand from left hand operand	<code>`expr \$a - \$b`</code>
* (Multiplication)	Multiplies values on either side of the operator	<code>`expr \$a * \$b`</code>
/ (Division)	Divides left hand operand by right hand operand	<code>`expr \$b / \$a`</code>
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	<code>`expr \$b % \$a`</code>
= (Assignment)	Assigns right operand in left operand	<code>a = \$b</code>
== (Equality)	Compares two numbers, if both are same then returns true.	<code>[\$a == \$b]</code>
!= (Not Equality)	Compares two numbers, if both are different then returns true.	<code>[\$a != \$b]</code>

- all the conditional expressions should be inside square braces with spaces around them,
- for example
- [\$a == \$b] is correct
- [\$a==\$b] is incorrect.

RELATIONAL OPERATORS

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[\$a -eq \$b]
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[\$a -ne \$b]
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[\$a -gt \$b]
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[\$a -lt \$b]
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[\$a -ge \$b]
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[\$a -le \$b]

BOOLEAN OPERATORS

x=5

y=10

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR . If one of the operands is true, then the condition becomes true.	[\$x -lt 10 -o \$y -gt 100] is true.
-a	This is logical AND . If both the operands are true, then the condition becomes true otherwise false.	[\$x -lt 20 -a \$y -gt 100] is false.

STRING OPERATORS

x="ab"

y="fg"

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[\$x = \$y] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[\$x != \$y] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z \$x] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n \$x] is not false.
str	Checks if str is not the empty string; if it is empty, then it returns false.	[\$x] is not false.

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/bash
```

```
read -p "enter your name: " first last
```

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

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

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:

```
read -p "Enter your password=" pass  
if test "$pass" == "admin"  
then  
echo "Password Verified"  
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 file
if [ ! -r $file ]
then
    echo "File is not read-able"
fi
```

EXAMPLE: FILE TESTING

```
#!/bin/bash

echo "Enter a filename: "
read file
if [[ ! -f $file || ! -r $file || ! -w $file ]]
then
    echo "File $file is not accessible"

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
```

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
```

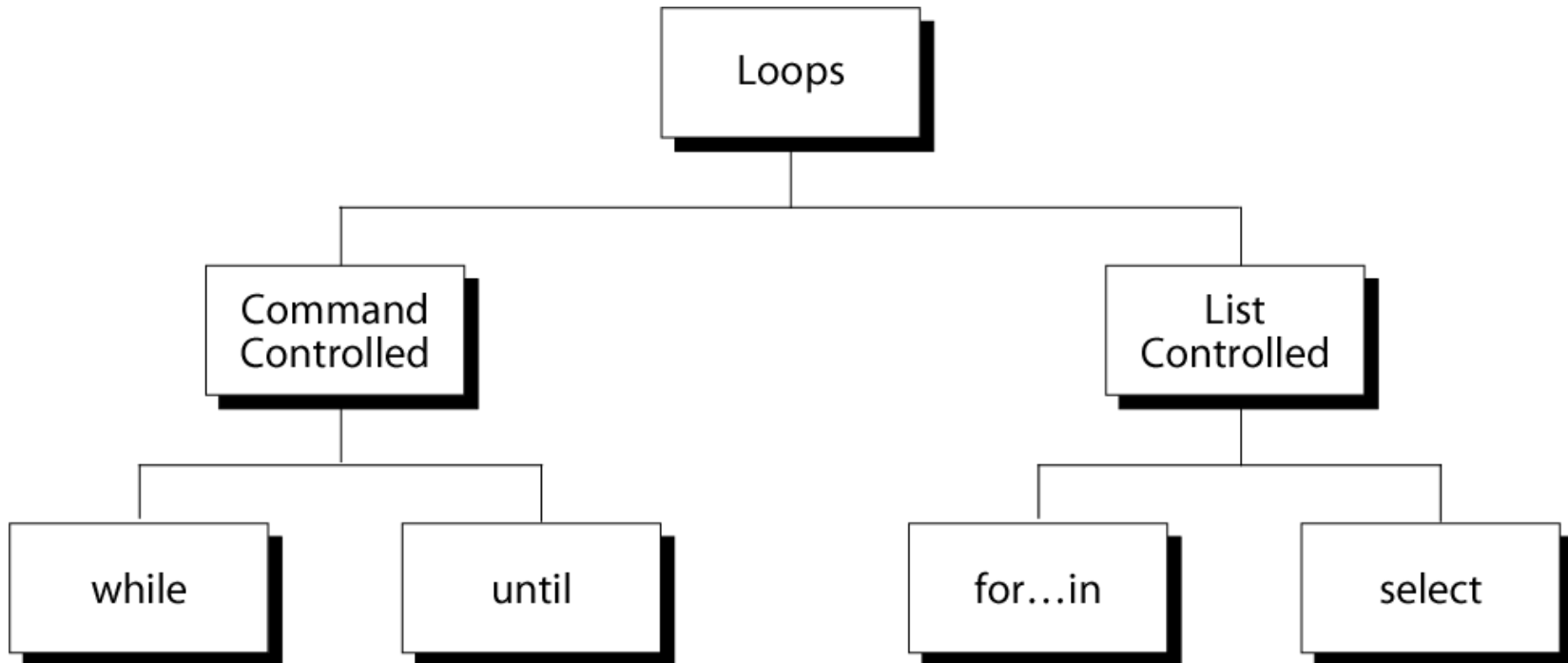
BASH PROGRAMMING: SO FAR

- Data structure
 - Variables
 - Numeric variables
 - Arrays
- User input
- Control structures
 - if-then-else
 - case

BASH PROGRAMMING: STILL TO COME

- Control structures
 - Repetition
 - do-while, repeat-until
 - for
 - select
- Functions
- Trapping signals

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
```

```
    COUNTER=`expr $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"
```


PRINT TABLE

```
read -p "Enter a number: " number

i=1

while [ $i -le 10 ]

do

    echo "$number x $i = $(($number*i))"

    i=$((i+1))

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
```

USING COMMA IN THE BASH C-STYLE FOR LOOP

```
#!/bin/bash
```

```
for ((i=1, j=10; i <= 5 ; i++, j=j+5))
```

```
do
```

```
echo "Number $i: $j"
```

```
done
```


PATTERN / NESTED FOR

```
Read -p "rows= " rows

for((i=1; i<=rows; i++))

do

    for((j=1; j<=i; j++))

        do

            echo -n "$j "

        done

    echo

done
```