LAB – 3: SHELL PROGRAMMING

Name: Shruti Sanjay Dhumal **PRN: 12111407**
Roll no.: 32 **Class: AI&DS-C**

Objectives:

1. To understand how to perform Shell programming in Unix/Linux.
2. To explain the purpose of shell programs.
3. Design and write shell programs of moderate complexity using variables, special variables, flow control mechanisms, operators, arithmetic and functions.

Description:

1. Shell
    - Interface to the user
    - Command interpreter
    - Programming features
2. Shell Scripting
    - Automating command execution
    - Batch processing of commands
    - Repetitive tasks
3. Shells in Linux
    - Many shells available
    - Examples -*sh, ksh, csh, bash*
    - Bash is most popular
4. The Bourne Again Shell
    - Abbreviated bash
    - Default in most Linux distributions
    - Most widely used on all UNIX platforms
    - Derives features from ksh, csh, sh, etc.
    - Support for many programming features variables, arrays, loops, decision
    - operators, functions, positional parameters

- Pipes, I/O re-direction

Shell variables:

1. System variables - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS
   - *BASH=/bin/bash* Our shell name
   - *HOME=/home/vivek* Our home directory
   - *LOGNAME=students* Our logging name
   - *PATH=/usr/bin:/sbin:/bin:/usr/sbin* Our path settings
   - *PS1=[\u@\h \W]\$* Our prompt settings
   - *PWD=/home/students/Common* Our current working directory
   - *SHELL=/bin/bash* Our shell name
   - *USERNAME=vivek* User name who is currently login to this PC
2. User defined variables (UDV) - Created and maintained by user. This type of variable defined in lower letters.
   - To define UDV use following syntax
     Syntax: *variable name=value*
     'value' is assigned to given 'variable name' and Value must be on right side = sign.
     Example:
     $ no=10 # this is ok
     $ 10=no # Error, NOT Ok, Value must be on right side of = sign.
   - To define variable called 'vech' having value Bus
     $ vech=Bus
   - To define variable called n having value 10
     $ n=10
   - To print or access UDV use following syntax
     Syntax: *$variablename*
     - echo Command: Use echo command to display text or value of variable.
     - Syntax: *echo [options] [string, variables...]*
     - Options
       a. *-n* Do not output the trailing new line.
       b. *-e* Enable interpretation of the following backslash escaped characters in the strings:
          I. \a alert (bell)

II. \b backspace
III. \c suppress trailing new line
IV. \n new line
V. \r carriage return
VI. \t horizontal tab
VII. \\\\ backslash

Shell Arithmetic:

- Use to perform arithmetic operations.
- Syntax: *expr op1 math-operator op2*
- Examples:
    - *$ expr 1 + 3*
    - *$ expr 2 - 1*
    - *$ expr 10 / 2*
    - *$ expr 20 % 3*
    - *$ expr 10 \* 3*
    - *$ echo `expr 6 + 3`*

The read Statement:

- Use to get input (data from user) from keyboard and store (data) to variable.
- Syntax: *read variable1, variable2,...variableN*
- Ex. *echo "Your first name please:"*
  read fname

Command line arguments:

- *$1,$2...$9* - positional parameter representing cnd line args
- *$#* - total no. of args
- *$0* - name of executed cmd
- *$** - complete set of positional parameters
- *$?* - exit status of last cmd
- *$$* - pid of current shell
- *$!* - pid of last background process

Control structures in shell:

1. **Decision control structure:**
   - The syntax is as follows:

     ```
     if <condition>
     then
             <do something>
     else
             <do something else>
     fi
     ```

   - Can use the 'test' or [ ] command for condition

     Ex.

     ```
     if test $a = $b                 if [$a = $b]
     then                            then
             echo $a                         echo $a
     fi                              fi
     ```

     Test & [ ] operator:

     [1] Numeric comparison
     - *-eq* equal to
     - *-ne* not equal to
     - *-gt* greater than
     - *-ge* greater than equal to
     - *-lt* less than
     - *-le* less than equal to

     [2] Compares two strings or a single one for a null value
     - *s1=s2* string s1 = s2
     - *s1!=s2* string s1 not equal to s2
     - *-n str* string str not a null string
     - *-z str* string str as a null string
     - *Stg string* str is assigned & not null
     - *s1==s2* string s1 = s2

     [3] Checks files attributes
     - *-f file* file exists & is regular
     - *-r file* file exists & is readable
     - *-w file* file exists & is writable
     - *-x file* file exists & is executable

        - *-d file* file exists & is directory
        - *-s file* file exists & has size greater than zero
        - *f1 –nt f2* f1 is newer than f2
        - *f1 –ot f2* f1 is older than f2
        - *f1 –ef f2* f1 is linked to f2

2. Loops
    1) for: The syntax is as follows:
```
for VAR in LIST
do
        <something>
done
```
    2) while: The syntax is as follows:
```
while <condition>
do
        <something>
done
```
    3) until : The syntax is as follows:
```
until <condition>
do
        <something>
done
```
3. The case Statement :The syntax is as follows:
```
case $variable-name in
pattern1) command....... command;;
pattern2) command....... command;;
patternN) command....... command;;
*) command....... command;;
esac
```

How to create file?

Syntax: *vi filename.sh*

- *i* - insert mode
- *:w* - save
- *:wq* - Save and quit the file

How to execute the shell script?

1. Method1

   $ chmod +x filename.sh

   $ ./filename.sh

2. Method 2

   $ bash filename.sh

How to de-bug the shell script?

While programming shell sometimes you need to find the errors (bugs) in shell script and correct the errors (remove errors - debug). For this purpose, you can use -v and -x option with sh or bash command to debug the shell script.

General syntax is as follows:

```
sh option { shell-script-name }
                 OR
bash option { shell-script-name }
```

Option can be

-v Print shell input lines as they are read.

-x After expanding each simple-command, bash displays the expanded value of PS4 system variable, followed by the command and its expanded arguments

Functions:

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles. Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions:

To declare a function, simply use the following syntax:

```
function_name () {
        list of commands
}
```

The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

Example:

Following is the simple example of using function:

```
#!/bin/sh
# Define your function here
Hello () {
        echo "Hello World"
}
# Invoke your function
Hello
```

When you would execute above script it would produce following result:

```
$./test.sh
Hello World
```

Passing argument to function in shell scripts:

Passing argument to the functions in shell script is very easy. Just use $1, $2, .. $n variables that represent arguments in the function. Following is the example of function that takes two arguments and prints them.

```
#!/bin/sh
# Define your function here
Hello () {
        echo "Hello World $1 $2"
}
# Invoke your function
Hello Zara Ali
```

This would produce following result:

```
$./test.sh
Hello World Zara Ali
$
```

Thus, if your function will use three arguments you can just use $1, $2 and $3 in your code.

You may want to use $# which gives you number of arguments passed to the function if you want to implement a function with variable arguments.

Returning Values from Functions:

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function. Based on the situation you can return any value from your function using the return command whose syntax is as follows:

```
return code
```

Here *code* can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

```
#!/bin/sh
# Define your function here
Hello () {
    echo "Hello World $1 $2"
    return 10
}
# Invoke your function
Hello Zara Ali
# Capture value returned by last command
ret=$?
echo "Return value is $ret"
```

This would produce following result:

```
$./test.sh
Hello World Zara Ali
Return value is 10
```

Nested Functions:

One of the more interesting features of functions is that they can call themselves as well as call other functions. A function that calls itself is known as a recursive function.

Following simple example demonstrates a nesting of two functions:

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

```
# Calling one function from another
number_one () {
    echo "This is the first function speaking..."
    number_two
}
number_two () {
    echo "This is now the second function speaking..."
}
# Calling function one.
number_one
```

This would produce following result:

```
This is the first function speaking...
This is now the second function speaking...
```

ALGORITHMS:

For Palindrome checking:

Steps:

1. Start.
2. Accept the string from user.
3. Find the actual length of string as len.
4. Initialize a pointer to character in string to 1 and also flag to true.
5. Take the character pointed to by the pointer and the character pointed to by len
6. If the characters are not found, make the flag as false and go to step 9.
7. Decrement variable len by 1 and increment pointer by 1.
8. If the value of len is less than or equal to 1 then repeat from step 5.
9. If the flag is true, display the message that ‘String is a palindrome’ else display ‘String is not palindrome.’
10. Stop.

Test Condition:

String should not be NULL.

For Bubble sort:

Steps:

1. Start.

2. Accept how many numbers to be sort say n .
3. Accept the numbers in array say num [].
4. Initialize i and j to 0.
5. While i<n .
6. do

```
        assign j=0
        while j<n
        do
                j=i
                if num[j]<num[i] then
                        swap num[i] and num[j]
                end if
        j=j+1
        done
        i=i+1
        done
```

7. Display the sorted list.
8. Stop.

Test Conditions:

1. A certain maximum number of elements can be entered.
2. Negative numbers are allowed.

For Substring checking:

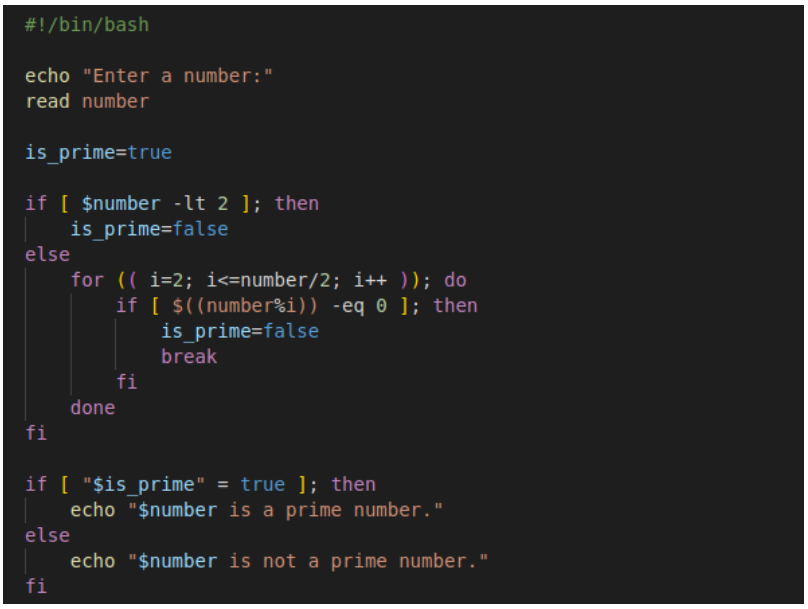
Steps:

1. Start.
2. Accept the two strings.
3. Compare two strings character by character.
4. If match is found then store the pointer of first string in an array.
5. Bring counter for second string to the first position.
6. Repeat the steps 3 to 5 until first string gets over.
7. Display the position array if substring exists else display substring does not exist.
8. Stop.

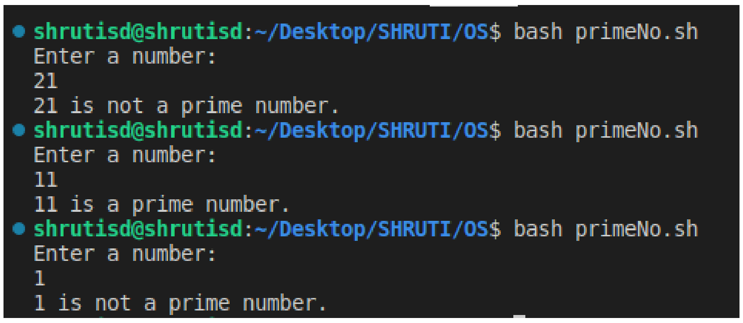
Test condition:

1. First string should not be NULL.

Prime Number:

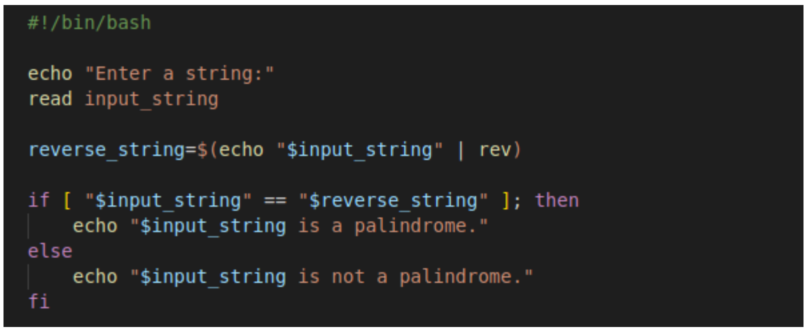


```
#!/bin/bash

echo "Enter a number:"
read number

is_prime=true

if [ $number -lt 2 ]; then
    is_prime=false
else
    for (( i=2; i<=number/2; i++ )); do
        if [ $((number%i)) -eq 0 ]; then
            is_prime=false
            break
        fi
    done
fi

if [ "$is_prime" = true ]; then
    echo "$number is a prime number."
else
    echo "$number is not a prime number."
fi
```

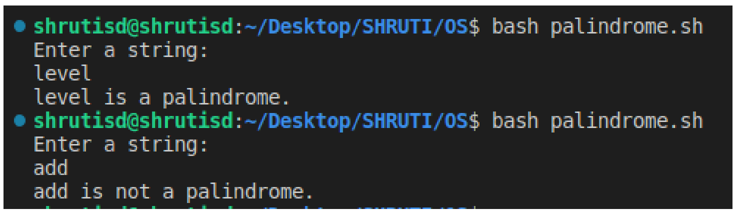


```
● shrutisd@shrutisd:~/Desktop/SHRUTI/OS$ bash primeNo.sh
  Enter a number:
  21
  21 is not a prime number.
● shrutisd@shrutisd:~/Desktop/SHRUTI/OS$ bash primeNo.sh
  Enter a number:
  11
  11 is a prime number.
● shrutisd@shrutisd:~/Desktop/SHRUTI/OS$ bash primeNo.sh
  Enter a number:
  1
  1 is not a prime number.
```

Palindrome:

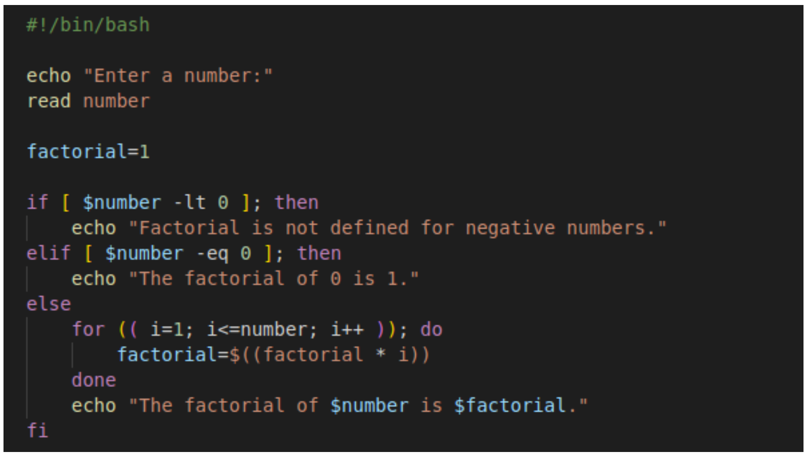


```
#!/bin/bash

echo "Enter a string:"
read input_string

reverse_string=$(echo "$input_string" | rev)

if [ "$input_string" == "$reverse_string" ]; then
    echo "$input_string is a palindrome."
else
    echo "$input_string is not a palindrome."
fi
```

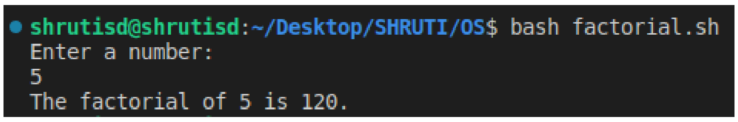


```
● shrutisd@shrutisd:~/Desktop/SHRUTI/OS$ bash palindrome.sh
  Enter a string:
  level
  level is a palindrome.
● shrutisd@shrutisd:~/Desktop/SHRUTI/OS$ bash palindrome.sh
  Enter a string:
  add
  add is not a palindrome.
```

Factorial:



```
#!/bin/bash

echo "Enter a number:"
read number

factorial=1

if [ $number -lt 0 ]; then
    echo "Factorial is not defined for negative numbers."
elif [ $number -eq 0 ]; then
    echo "The factorial of 0 is 1."
else
    for (( i=1; i<=number; i++ )); do
        factorial=$((factorial * i))
    done
    echo "The factorial of $number is $factorial."
fi
```



```
shrutisd@shrutisd:~/Desktop/SHRUTI/OS$ bash factorial.sh
Enter a number:
5
The factorial of 5 is 120.
```

Conclusion: Thus, we have studied how to use shell script. It is useful for integrating our existing applications and useful for system administrator.

FAQs:

1. Define shell. What are types of shell?
2. What are different control structures used in shell programming?
3. What do you mean by positional parameters & enlist them?
4. How to specify default statement in case statement?
5. What are types of variables?
6. How functions are used in Shell scripting?