“Lab Report- 03”

Course Code:	Course Title:
CSE 324	Operating System Lab

Lab Report Details	
Experiment No	: 03
Experiment Name	: Shell Programming with Function.
Lab Perform Date	: 00-10-2024
Report Submission Date	: 30-10-2024

Submitted To:	Submitted By:
Faiza Feroz Lecturer Department of **“CSE”** Daffodil International University	Name : Munna Biswas ID : 221-15-5261 Section : 61_J2 Department of “CSE”

Daffodil International University.

Title:

Shell Programming with Function.

Equipment:

- **Hardware Specifications**:
  - **Processor**: 11th Gen Intel® Core™ i3-1115G4 @ 3.00GHz.
  - **Memory (RAM)**: 8 GB.
  - **Storage**: 512 GB SSD.
  - **Architecture**: 64-bit.
- **Software Specifications**:
  - **Operating System**: Ubuntu 22.04 LTS in Virtual machine.
  - **Kernel Version**: 6.8.0-41-generic.
  - **Shell Version**: Bash 5.1.16(1)

Experimental steps:

1. How to Write a Function in Shell Script

In shell scripting, a function is a reusable block of code that performs a specific task. Functions allow us to break down complex tasks into smaller, manageable parts and help to avoid code duplication. Once a function is defined, it can be called or invoked by its name multiple times within the script.

In shell scripting, a function is defined as:

```
my_function() {
    echo "This is a simple function."
}
my_function  # Calling the function
```

2. How to Pass Parameters

Functions in shell scripts can accept parameters, which are passed to them when they are called. These parameters make functions more versatile, allowing them to work with different inputs. In shell scripting, parameters passed to a function are accessed by position, like $1, $2, etc., where $1 is the first parameter, $2 is the second, and so on.

Parameters are passed by simply listing them after the function name:

```
greet() {
    echo "Hello, $1!"
}
greet "Munna"  # Output:Hello, Munna!
```

3. How to Get a Return Value of the Function

In shell scripting, functions don't directly return values like functions in other programming languages. Instead, they can:

- Return an exit status using the return command, where 0 usually means success, and any non-zero value indicates failure.
- Alternatively, functions can store their output in a variable, which can then be accessed after the function call.

Using the return command with an integer value:

- return <number> is typically used to signal success (return 0) or an error code (non-zero).

Using variables:

- Instead of return, we can set a global variable within the function to hold a result.

We can use return to return an exit status or set a global variable to store a result.

```
square() {
    result=$(( $1 * $1 ))
}
square 4
echo "Square: $result"    # Output: Square: 16
```

4. How to Write Recursive Functions

Important Points about Recursion in Shell Scripts:

1. **Base Condition**: Every recursive function must have a base condition to stop the recursion. Without this, the function will keep calling itself indefinitely, leading to a stack overflow.
2. **Self-Call**: The function calls itself with a modified argument (usually a reduced value) to approach the base condition with each recursive call.
3. **Memory and Performance**: Recursive functions in shell scripts can be memory-intensive and slow for complex problems because each function call creates a new layer on the stack.
4. **Calculating Factorial Using Recursion** To demonstrate recursion, let's write a shell script function that calculates the factorial of a number using recursion.

Here's an example of a recursive function for calculating factorials.

```
factorial() {
    if [ $1 -le 1 ]; then
        echo 1
    else
        prev=$(factorial $(( $1 - 1 )))
        echo $(( $1 * prev ))
    fi
}
```

```
echo "Factorial of 5 is $(factorial 5)"
```

5. Function to Swap Two Numbers

```
#Swap two numbers:
swap() {
        local temp=$1
        set -- "$2" "$temp"
        echo "After Swap the number is: $1 and $2"
}
swap 3 7
echo
```

Output is:

```
cyberseeker@cyberseeker-VirtualBox:~$ nano lab3.sh
cyberseeker@cyberseeker-VirtualBox:~$ chmod +x lab3.sh
cyberseeker@cyberseeker-VirtualBox:~$ ./lab3.sh
This is our Labreport-3
cyberseeker@cyberseeker-VirtualBox:~$ nano lab3.sh
cyberseeker@cyberseeker-VirtualBox:~$ ./lab3.sh
This is our Labreport-3

After Swap: 7 and 3

cyberseeker@cyberseeker-VirtualBox:~$ nano lab3.sh
cyberseeker@cyberseeker-VirtualBox:~$ ./lab3.sh
This is our Labreport-3

After Swap the number is: 7 and 3
```

6. Recursive Function to Find Factorial

```
#Find factorial Numbers:
factorial() {
        if [ $1 -le 1 ]; then
                echo 1
        else
                prev=$(factorial $(( $1 - 1 )))
                echo $(( $1 * prev ))
        fi
}
echo "Factorial of 5 is $(factorial 5)"
echo
```

Output is:

```
cyberseeker@cyberseeker-VirtualBox:~$ nano lab3.sh
cyberseeker@cyberseeker-VirtualBox:~$ ./lab3.sh
This is our Labreport-3

After Swap the number is: 7 and 3

Factorial of 5 is 120
```

7. Recursive Function to Find Fibonacci Series Up to N

```
#Find fibonacci series:
echo "Fibonacci Numbers are: "
fibonacci() {
        if [ $1 -le 0 ]; then
                echo 0
        elif [ $1 -eq 1 ]; then
                echo 1
        else
                echo $(( $(fibonacci $(( $1 - 1 ))) + $(fibonacci $(( $1 - 2 ))) ))
        fi
}

for i in $(seq 0 5); do
        echo "$(fibonacci $i)"
done
echo
```

Output is:

```
Fibonacci Numbers are:
0
1
1
2
3
5
```

Home Task

1. Declare and Access Arrays

```
my_array=(10 20 30 40)
echo ${my_array[0]}   # Output: 10
```

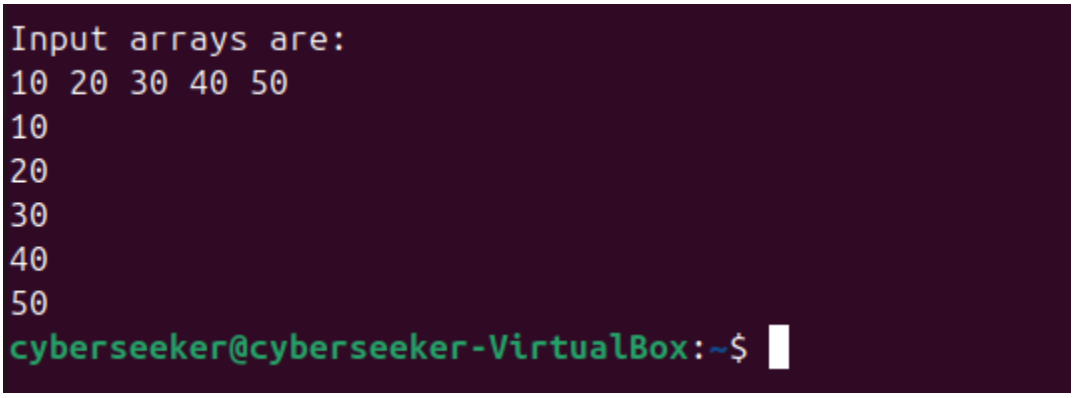
2. Access Specific Elements of Arrays

```
echo ${my_array[2]}   # Output: 30
```

3. Input an Array, Print It, and Pass to Function

```
#Input an array:
echo "Input arrays are: "
read -a my_array
print_array() {
        for value in "$@"; do
                echo $value
        done
}
print_array "${my_array[@]}"
```

Output is:



Results

1. **Successful Function Execution**:
    - Demonstrated the successful creation and execution of functions in shell scripts for specific tasks, such as greeting users and calculating squares of numbers.
    - Showcased how parameters are passed to functions and how they modify the behavior based on input values.
2. **Parameter Passing and Return Values**:
    - Verified that parameters passed to functions are correctly interpreted within the function scope, enabling flexible function behavior.
3. **Recursive Function Results**:
    - The factorial function and Fibonacci function worked as expected, with recursive calls calculating the values step-by-step until reaching the base condition.
4. **Array Input and Manipulation**:
    - Successfully captured user input to form an array, accessed specific elements, and passed the array as a parameter to a function.
    - Demonstrated array manipulation, which can be complex in shell scripting but was achieved through direct indexing and looping.

Discussion

1. **Understanding Function Basics**:
    - Shell scripting functions provide a structured way to reuse code, enhancing script organization. Although simpler than functions in many programming languages, shell functions are versatile and can handle a wide range of tasks when used effectively.
2. **Advantages and Limitations of Parameter Passing**:
    - Passing parameters to shell functions allows for dynamic operations based on input, making scripts adaptable. However, shell scripting limits parameter handling compared to languages like Python or Java, as it uses positional parameters ($1, $2, etc.) without named arguments.
3. **Recursive Functions in Shell Scripting**:
    - While recursion in shell scripts is possible, it is generally not as efficient as in other programming languages. Shell scripts lack the memory optimization for recursion found in languages like C or Python, which can lead to stack overflow errors with high recursion depths. In practice, iterative solutions are often preferable in shell scripts.
4. **Array Handling and Limitations**:
    - Array handling in shell scripting can be challenging, as arrays are limited in functionality compared to languages designed for complex data handling. Bash arrays do not support multi-dimensional structures directly, making complex data organization cumbersome.
5. **Recommendations for Optimizing Shell Scripts**:
    - To optimize shell scripts, avoid heavy recursion and extensive array manipulations whenever possible.
    - Consider using loops instead of recursion, especially for larger datasets or deeper recursive calls.