Linux Programming: Assignment-8

Samriddhi Guha | ENG24CY0157 | 3-C (25)

11. What is a user-defined function in shell scripting? Explain with an example. (CO4)

In shell scripts, a user-defined function can be considered as a reusable block of code to accomplish a particular modular task. Functions allow an extensive shell script to be broken down into smaller, manageable, and organized logic sections enhancing code readability and eliminating code duplication.

Function Declaration Syntax:

function function_name { commands; }

function_name () { commands; }

Function Arguments:

Arguments are positional when passed into a function, and can be accessed in the function with special variables:

$1, $2, $3, etc.: positional arguments

$#: the number of arguments passed, $@: all the arguments passed in.

Example:

```
#!/bin/bash
# Function to greet a user by name

greet_user () {
   # Check if a name was provided (expecting 1 argument)
   if [ "$#" -ne 1 ]; then
      echo "Error: Please provide exactly one name."
      return 1
   fi

   # Access the first argument using $1
   echo "Hello, $1! Welcome to the world of shell scripting."
   return 0
}

# Call the function with an argument
greet_user "Samriddhi"

# Call the function without an argument to see the error handling
greet_user
```

12. Write a bash script with a function that multiply two integer numbers. (CO4)

This script defines a function that accepts two numbers as arguments and uses the arithmetic expansion $(( )) to perform the multiplication.

```
#!/bin/bash
# Script with a function to multiply two integer numbers

# Function definition
multiply () {
   # Check if exactly two arguments were provided
   if [ "$#" -ne 2 ]; then
      echo "Error: Two numbers are required for multiplication."
      return 1
   fi

   NUM1=$1
   NUM2=$2

   # Arithmetic expansion for multiplication
   RESULT=$(( NUM1 * NUM2 ))

   echo "The product of $NUM1 and $NUM2 is: $RESULT"
   return 0
}

# Main script execution
echo "--- Integer Multiplier ---"

# Call the function with two numbers
multiply 15 7

# Example of another call
multiply 9 11

echo "--------------------------"
```

13. Explain how arrays (1D, 2D, and 3D) are declared in bash scripting. (CO4)

Bash scripting primarily supports one-dimensional (1D) arrays that are indexed starting from . Although multi-dimensional arrays (2D and 3D) are not natively supported, they can be emulated through clever indexing - or perhaps associative arrays.

1D (One-Dimensional) Array:

- Purpose: The default array type - a basic list of values.
- Declaration: Either explicitly through declare -a or implicitly by assigning values in parentheses.
- Example:

  declare -a COLORS=("Red" "Green" "Blue")
  # or: CITIES= ("Delhi" "Mumbai" "Kolkata")
- Access: Accessing an element involves using the bracketed index in this way: ${COLORS[0]} returns "Red".

2D and 3D Arrays (Simulated):

- Purpose: To represent data in a table (rows and columns).
- Declaration: Since Bash does not allow for nested arrays (unlike other languages), the common way to emulate a 2D or 3D array is to use one-dimension arrays with a flattened index, or associative arrays.
- Example:

  # 2x2 matrix simulation
  MATRIX=(1 2 3 4)
  *# Access element at (row 1, col 0) where N_COLUMNS=2: index = 1*2 + 0 = 2*
  echo ${MATRIX[2]} # Outputs '3'

14. Write a shell script to display elements of an array. (CO4)

This script demonstrates declaring an array and accessing all its elements using different methods.

```
#!/bin/bash
# Script to declare and display all elements of an array

# Declare and initialize a 1D array
FRUITS=("Apple" "Banana" "Cherry" "Grape")

echo "--- Displaying Array Elements ---"

# 1. Display all elements at once
echo "All elements: ${FRUITS[@]}"

echo "---------------------------------"

# 2. Display elements using a for loop (standard iteration)
echo "Elements using a For Loop:"
for fruit in "${FRUITS[@]}"; do
    echo "  -> $fruit"
done

echo "---------------------------------"
```

```
# 3. Display elements using a C-style for loop (index iteration)
echo "Elements using a C-style For Loop (with Index):"
NUM_ELEMENTS=${#FRUITS[@]}

for (( i=0; i<NUM_ELEMENTS; i++ )); do
    echo "  Index $i: ${FRUITS[i]}"
done
```

15. What is the purpose of cron in Linux? (CO4)

The purpose of *cron* in Linux is to allow commands or scripts to run automatically, at specified intervals. It is the main utility for task automation and maintenance of the system.

- How it works - cron jobs are configured in a file often called a crontab (cron table). Each line in the crontab defines a job in a specific time format (minute, hour, day, month, day of the week) then the command to run follows the format.
- Use cases - Automating tasks like backing up the system, rotating log files, checking the system status, sending reports on a scheduled basis, and running a cleanup script.
- The daemon - The cron daemon cron runs in the background always. This daemon checks the crontabs every minute to see if it is time to run the job.

16. Write a cron job to run a backup script every day at midnight. (CO4)

To execute a backup script, /usr/local/bin/daily_backup.sh, daily, exactly at midnight (12:00 AM), the corresponding crontab entry would be:

Crontab Format: *minute hour day_of_month month day_of_week command_to_execute*

Example:
*0 0 * * * /usr/local/bin/daily_backup.sh*

Here,

- 0 (minute): Run when the minute is 0 (beginning of the hour).
- 0 (hour): Run when the hour is 0 (midnight).
- * (day_of_month): Run every day of every month.
- * (month): Run every month.
- * (day_of_week): Run every day of the week.

17. How do you schedule a one-time job using at command? (CO4)

To plan a one-time job using the at command in Linux, we can follow these steps:

Start the at command with the time to execute:

at [time] [date]
For example, if you want to schedule a job for today at 3:00 PM:
at 3:00 PM

Or, we can schedule a job for 10:30 AM on December 25th:
Code example

```
at 10:30 AM Dec 25
```

Relative times can also be used, i.e,

Example:

at now + 5 minutes
at tomorrow
at next monday

and then, enter the commands that you want to executed.
After entering the at command and the time, the at utility will give you a prompt (i.e., at>). At this prompt, type in the commands or script you would like to have executed, pressing enter after each command, just as you would normally. After you are finished entering your job, exit and save the job.

Once you have entered all of your commands, you can press Ctrl+D, and this will cause the prompt to enter saved mode, while exiting the at prompt. The system will then display the job number, and the net time that was set.

Example

```
at 17:00 tomorrow
at> echo "Hello from the future!" > /tmp/future_message.txt
at> ^D  # Ctrl+D to exit and save the job.
job 1 at Sat Oct 11 13:00:00 2025
```

18. Write a script to display disk usage using df and du. (CO4)

This script uses *df* to show filesystem-level usage and du to show disk space used by a specific directory, both using human-readable formats.

```
#!/bin/bash
# Script to display overall and specific disk usage

# 1. Display Filesystem Disk Usage (df)
```

```
echo "--- Overall Filesystem Disk Usage (df) ---"
# -h: human-readable format
# -T: print filesystem type
df -hT

echo "------------------------------------------------"

# 2. Display Detailed Directory Disk Usage (du)
# Use the root directory for demonstration
TARGET_DIR="/home/samriddhi"

echo "--- Disk Usage for Directory: $TARGET_DIR (du) ---"
# -s: summarize (show only a total)
# -h: human-readable format
du -sh "$TARGET_DIR"

echo "------------------------------------------------"
```

19. How can you log the output of a script using the tee command? (CO4)

The *tee* command is used in a script pipeline to simultaneously display the output on the terminal (Standard Output) *and* write a copy of that output to a log file.

Method: Pipe the script's output to the tee command.

- Syntax: script_command | tee [OPTIONS] log_file

Example:

```
#!/bin/bash
# Script demonstrating output logging with tee

LOG_FILE="script_run.log"

echo "Starting script execution..."

# Redirect all output (including from 'find') to tee
# -a: Appends to the log file instead of overwriting (recommended for logging)
{
   echo "Current time: $(date)"
   echo "Finding configuration files..."
   find /etc -maxdepth 1 -name "*.conf" 2>/dev/null  # Example command
   echo "Task complete."
} | tee -a "$LOG_FILE"

echo "------------------------------------------------------"
echo "Script finished. Check terminal and log file: $LOG_FILE"
```

20. Explain with an example how shell scripting can automate system administration tasks. (CO4)

Shell scripting automates system administration by combining multiple commands into a single, executable file, allowing complex, repetitive, or time-sensitive tasks to be run automatically and reliably.

Explanation:

Automation is achieved by:

1. Sequencing Tasks: Ensuring commands run in the correct order.
2. Conditional Logic: Using if/else statements to react to system conditions (e.g., checking if a service is running).
3. Scheduling: Integrating with cron to run the script automatically.

Example:

```
#!/bin/bash

# Automated system health checker

# Define the threshold (e.g., 90% usage)
DISK_THRESHOLD=90
LOG_FILE="/var/log/health_check.log"

# Get the current disk usage percentage for the root partition
CURRENT_USAGE=$(df -h / | awk 'NR==2 {print $5}' | sed 's/%//')

echo "Health check started at $(date)" | tee -a "$LOG_FILE"

# Check if disk usage exceeds the threshold
if [ "$CURRENT_USAGE" -ge "$DISK_THRESHOLD" ]; then
    ALERT_MESSAGE="CRITICAL ALERT: Root disk usage is at $CURRENT_USAGE%."

    # 1. Log the alert
    echo "$ALERT_MESSAGE" | tee -a "$LOG_FILE"

    # 2. Automate an action (e.g., sending an email or shutting down a service)
    # mail -s "Disk Alert" admin@example.com <<< "$ALERT_MESSAGE"

else
    echo "INFO: Disk usage is acceptable at $CURRENT_USAGE%." | tee -a "$LOG_FILE"
fi
```