4th SEMINAR – OPTIONS, SWITCHES & MANAGING INPUT/OUTPUT (I/O)

(8 sections, total 40-50 min)

WELCOME AND OBJECTIVES:

Good day, everyone. Today's seminar is meticulously designed to cater to a wide range of skill levels, with the primary requirement being a basic understanding of the Linux terminal environment. Our journey through the world of Bash scripting continues with an exploration of how to enrich our scripts with interactive elements, secure data handling practices, and sophisticated input and output management strategies. The seminar will unfold through the following structured sections:

1. **Enhancing Bash Scripts with Command-Line Options and Switches**: We'll kickstart our seminar with a deep dive into the world of command-line options and switches. This section is all about amplifying the interactivity of your bash scripts, allowing them to dynamically process data provided by users.
2. **Shift Command and Command-Line Switches**: Following up, we'll explore the shift command alongside command-line switches. These concepts are pivotal in enhancing the flexibility and user interaction of your scripts, enabling them to handle a variable number of arguments and providing structured options for users.
3. **Distinguishing Between Keys and Parameters**: Our third segment focuses on distinguishing between command-line options (or keys) and parameters, a common scenario in bash scripting. We'll also delve into handling switches with values and implementing standard key conventions to boost the usability and interaction of your scripts.
4. **Secure and Efficient Data Handling in Bash Scripts**: In this section, we address advanced techniques for managing sensitive user input and reading data from files efficiently. These practices are paramount in enhancing the security and functionality of your bash scripts, especially in contexts that demand confidentiality and precise data processing.
5. **Understanding and Managing Input and Output in Bash Scripts**: Here, we'll examine the fundamental concepts of handling input and output streams in Linux environments. Mastery of these concepts is crucial for effectively directing script interactions, ensuring accurate data processing and logging, and maintaining confidentiality when necessary.
6. **Advanced Techniques in Bash Scripting: Redirecting Output**: As we progress, we'll delve into the subtleties of output redirection within bash scripts. This ability is key to controlling where your script's output is directed, be it for logging purposes, error tracking, or data processing.
7. **Mastering Input Redirection in Bash Scripts**: Our penultimate section explores the intricacies of input redirection. This mechanism is indispensable in scenarios where scripts need to read data from files or other sources rather than the standard input (STDIN).
8. **File Descriptor Management and Output Suppression in Bash Scripts**: Finally, we will tackle advanced topics such as managing file descriptors effectively and suppressing unnecessary output. These skills are vital for maintaining script efficiency, ensuring data security, and controlling the interaction of your script with the user and the system.

By the end of today's session, you will be equipped with a deeper understanding and practical skills to create more dynamic, secure, and efficient bash scripts.

4.1 Understanding Command-Line Parameters:

1. **Basics of Command-Line Parameters**:
    - ***Definition:*** Command-line parameters allow users to pass information to scripts at the time of execution.
    - ***Usage Example:*** Running a script with parameters ./myscript 10 20 passes "10" and "20" as arguments to the script.
2. **Positional Parameters**:
    - Mechanism: Bash automatically assigns passed arguments to variables named $1, $2, ..., $9.
    - Special Variable $0: Represents the script's name.
    - Accessing Parameters: Within the script, refer to these variables directly to use the passed values.

PRACTICAL EXAMPLES AND EXPECTED OUTPUTS:

1. **Displaying Script Name and Parameters**:
    - Script Content:

```
#!/bin/bash
echo $0  # Displays the script's name
echo $1  # Displays the first parameter
echo $2  # Displays the second parameter
echo $3  # Displays the third parameter
```

    - Execution and Output:

```
./myscript 5 10 15

5

10

15
```

2. **Calculating Sum of Parameters:**
    - Script Functionality: Adds two numbers passed as parameters.
    - Script Snippet:

```
#!/bin/bash
total=$[ $1 + $2 ]
echo "The first parameter is $1."
echo "The second parameter is $2."
echo "The sum is $total."
```

    - Execution and Verification:

```
./myscript 5 10
```

```
The first parameter is 5.

The second parameter is 10.

The sum is 15.
```

3. **Handling String Parameters:**
    - Script Purpose: Display a personalized greeting using a passed string.
    - Script Content:

```
#!/bin/bash
echo "Hello $1, how do you do"
```

    - Execution and Output:

```
./myscript Gogu

Hello Gogu, how do you do
```

SPECIAL CONSIDERATIONS:

1. **Handling Spaces in Parameters**:
    - Use quotes to enclose parameters containing spaces to treat them as single entities.
2. **Accessing Beyond Ninth Parameter**:
    - Utilize curly braces to reference parameters beyond the ninth, e.g., ${10}.

4.2a Understanding the Shift Command:

1. **Basics of the Shift Command**:
    - ***Definition:*** The shift command in bash scripts shifts positional parameters to the left, effectively decreasing the number of parameters and changing their position.
    - ***Mechanism:*** After executing shift, $2 becomes $1, $3 becomes $2, and so on, while the original $1 is discarded.
2. **Practical Example and Expected Output**:
    - Script Utilization: A loop iterates over all parameters, echoing each one before shifting.
    - Script Content:

```
#!/bin/bash
count=1
while [ -n "$1" ]; do
  echo "Parameter #$count = $1"
```

```
  count=$(( count + 1 ))
  shift
done
```

- Execution and Analysis: Command: ./myscript 1 2 3 4 5 Expected Output:

```
./myscript 1 2 3 4 5

Parameter #1 = 1

Parameter #2 = 2

Parameter #3 = 3

Parameter #4 = 4

Parameter #5 = 5
```

- Discussion: This example illustrates how shift can be used to process each parameter sequentially.

4.2b Command-Line Switches:

1. **Introduction to Command-Line Switches**:
   - ***Definition:*** Command-line switches, typically prefixed with a dash (**-**), allow users to specify options or flags that modify the script's behavior.
   - ***Usage:*** They are processed typically in a while loop combined with a case statement for handling different options.
2. **Script Example and Expected Behavior**:
   - Script Details: The script evaluates command-line switches using case statements.
   - Script Snippet:

```
#!/bin/bash
echo
while [ -n "$1" ]; do
  case "$1" in
    -a) echo "Found the -a option";;
    -b) echo "Found the -b option";;
    -c) echo "Found the -c option";;
    *)  echo "$1 is not an option";;
  esac
  shift
done
```

   - Execution and Expected Output: Command: ./myscript -a -b -c -d Expected Console Output:

```
./myscript -a -b -c -d

Found the -a option

Found the -b option

Found the -c option
```

```
-d is not an option
```

Analysis: This demonstrates how scripts can handle different switches and provide feedback to the user.

4.3 Distinguishing Between Keys and Parameters:

1. **Separating Options from Parameters**:
    - ***Mechanism:*** The double dash (--) serves as a signal in the script to distinguish between options and trailing parameters.
    - Script Illustration:

```
#!/bin/bash
while [ -n "$1" ]; do
  case "$1" in
    -a) echo "Found the -a option" ;;
    -b) echo "Found the -b option";;
    -c) echo "Found the -c option" ;;
    --) shift; break ;;
    *)  echo "$1 is not an option";;
  esac
  shift
done
count=1
for param in "$@"; do
  echo "Parameter #$count: $param"
  count=$((count + 1))
done
```

    - Expected Output:

```
./myscript -a -b -c -- 5 10 15

Found the -a option

Found the -b option

Found the -c option

Parameter #1: 5

Parameter #2: 10

Parameter #3: 15
```

2. **Handling Keys with Values:**
    - Implementation: Introduce switch parameters that accept additional data.
    - Script Example:

```
#!/bin/bash
while [ -n "$1" ]; do
  case "$1" in
    -a) echo "Found the -a option";;
```

```
    -b) param="$2"
        echo "Found the -b option, with parameter value $param"
        shift ;;
    -c) echo "Found the -c option";;
    --) shift; break ;;
    *)  echo "$1 is not an option";;
  esac
  shift
done
count=1
for param in "$@"; do
  echo "Parameter #$count: $param"
  count=$((count + 1))
done
```

- Expected Behavior:

```
./myscript -a -b 15 -d

Found the -a option

Found the -b option, with parameter value 15

-d is not an option
```

Adhering to Standard Keys:

In the Linux ecosystem, certain command-line switches have standardized meanings, which can enhance script familiarity and user-friendliness. Examples include:

- **-a**: List all objects
- **-c**: Make a count
- **-d**: Specify directory
- **-h**: Display command help

Utilizing these standard switches in your scripts can provide a more intuitive interface for users accustomed to Linux conventions.

Enhanced User Data Reception:

Beyond static command-line data, scripts often need dynamic input from users during execution. The read command facilitates this interaction, enabling scripts to prompt for and capture user input directly.

1. **Basic User Input**:
   - Script Snippet:

```
#!/bin/bash
echo -n "Enter your name: "
read name
echo "Hello $name, welcome to my program."
```

2. **Timed Input Collection:**
    - Implementation: The read command can be paired with the -t option to limit input wait time.
    - Script Example:

```
#!/bin/bash
if read -t 5 -p "Enter your name: " name; then
  echo "Hello $name, welcome to my script"
else
  echo "Sorry, too slow!"
fi
```

This allows scripts to proceed even in the absence of user interaction within the specified timeframe.

4.4 Entering Passwords Securely:

1. **Concealing User Input**:
    - ***Context:*** Sensitive data, such as passwords, should not be visible on-screen when entered.
    - ***Implementation:*** Utilize the **read** command with the **-s** option to hide user input.
    - Script Example:

```
#!/bin/bash
read -s -p "Enter your password: " pass
echo -e "\nIs your password really $pass?"
```

    - Expected Behavior: The script prompts the user to enter a password, conceals the input, and then, for demonstration purposes (though not recommended in real scenarios), echoes it back to ensure it was captured correctly.
2. **Practical Considerations**:
    - Security Note: Directly echoing passwords, as shown, is usually not advised outside of a teaching context due to privacy concerns.

Reading Data from a File:

1. **Efficient File Processing**:
    - Objective: Read and process each line of a text file within a script.
    - Basic Script Structure:

```
#!/bin/bash
count=1
cat myfile | while read line; do
  echo "Line $count: $line"
  count=$((count + 1))
done
echo "Finished"
```

- Expected Output: The script reads lines from 'myfile', numbers them, and displays each on the console.

2. **Best Practices and Alternatives**:
   - Discussion: While the use of cat followed by a pipe into while read is intuitive for beginners, there are more efficient methods, such as using redirection or looping directly over the read command, which avoid unnecessary subprocesses.

4.5 Standard File Descriptors in Linux:

1. **Foundation of I/O Streams**:
   - Linux treats nearly everything as files, utilizing file descriptors to manage data streams.
   - **Standard Descriptors**:
     - **STDIN (0)**: Standard Input, typically the keyboard.
     - **STDOUT (1)**: Standard Output, typically the screen.
     - **STDERR (2)**: Standard Error, for error messages, also typically the screen.

2. **Understanding STDIN**:
   - The default input source for scripts and commands, usually the keyboard.
   - Input redirection (<) allows scripts to take input from files instead of the keyboard.
   - Example:

```
cat < myfile
```

This redirects the contents of 'myfile' into the cat command, simulating keyboard input.

3. Exploring STDOUT:

- The default output destination for scripts and commands.
- Output redirection (> and >>) directs the script's output to files.
- Example:

```
echo "Hello World" > myfile
```

This command writes "Hello World" into 'myfile', overwriting existing content.

PRACTICAL EXAMPLES AND EXPECTED OUTPUTS:

1. **Redirecting STDOUT**:
    - Appending data to a file:

```
pwd >> myfile
```

Expected Behavior: The current directory path is appended to 'myfile' without erasing its previous content.

2. **Handling STDERR:**

- Understanding that by default, STDERR outputs to the screen just like STDOUT.
- Redirecting STDERR:

```
ls -l nonexistentfile 2> errorlog
```

Expected Behavior: Error messages are redirected to 'errorlog', not displayed on screen.

ADVANCED REDIRECTION TECHNIQUES:

1. **Separating Output and Error Streams**:
    - Directing output and errors to different files:

```
ls -l validfile nonexistentfile 1> outputlog 2> errorlog
```

This separates successful command output from error messages, aiding in debugging and logging.

2. **Unified Redirection:**

- Merging STDOUT and STDERR into a single file:

```
ls -l validfile nonexistentfile &> combinedlog
```

This approach is useful for capturing all script interactions in a single log for review.

4.6 Temporary vs. Permanent Redirection:

1. **Temporary Output Redirection**:
    - This approach redirects the output of a single command or line within a script.
    - Example and Usage:

```
#!/bin/bash
echo "This is an error" >&2
echo "This is normal output"
```

    - Running ./myscript 2> content redirects only STDERR to 'content', leaving STDOUT to be displayed on the console.

- Expected Behavior: "This is an error" is written to the file 'content', while "This is normal output" is displayed on the screen.

2. **Permanent Output Redirection:**
   - Applies to all or part of a script's output, changing the destination of STDOUT or STDERR globally within the script.
   - Utilizing exec:

```
#!/bin/bash
exec 1>outfile
echo "This is a test of redirecting all output"
echo "from a shell script to another file."
echo "without having to redirect every line"
```

   - Expected Outcome: All echoed text is written to 'outfile', none is displayed on the console.

SCRIPT EXAMPLES AND EXPECTED OUTPUTS:

1. **Combining Temporary and Permanent Redirections**:
   - Script Structure:

```
#!/bin/bash
exec 2>myerror
echo "This is the start of the script"
echo "now redirecting all output to another location"
exec 1>myfile
echo "This should go to the myfile file"
echo "and this should go to the myerror file" >&2
```

   - This script demonstrates dynamic redirection changes within execution: STDOUT initially goes to the console, then to 'myfile', while STDERR is directed to 'myerror'.
   - Expected Outputs:
     - 'myerror' contains "and this should go to the myerror file".
     - 'myfile' contains "This should go to the myfile file".

DETAILED DISCUSSION AND CONSIDERATIONS:

1. **Understanding Redirection Commands**:
   - >&2 redirects STDOUT to STDERR.
   - exec 1>file changes the destination of STDOUT for the duration of the script or until further redirection.
   - exec 2>file changes the destination of STDERR in a similar manner.
2. **Practical Applications**:
   - Logging: Separating standard and error outputs into different files for easier debugging.
   - Data Handling: Directing output to files for later processing or archiving.

4.7 Concepts and Implementation:

1. **Basics of Input Redirection**:
    - Input redirection allows a script to read input from a file or another source instead of the keyboard.
    - Executing ***exec 0< myfile*** changes the source of STDIN to 'myfile'.
2. **Practical Example of Input Redirection**:
    - Example Script:

```
#!/bin/bash
exec 0< testfile
count=1
while read line; do
  echo "Line #$count: $line"
  count=$((count + 1))
done
```

Expected Output: Reads lines from 'testfile' and prints them, numbering each line.

ADVANCED TECHNIQUES AND CUSTOM DESCRIPTORS:

1. **Creating Custom Output Redirection**:
    - Beyond the standard descriptors, scripts can utilize additional ones (3-8) for specific tasks.
    - Example:

```
#!/bin/bash
exec 3>myfile
echo "This should display on the screen"
echo "and this should be stored in the file" >&3
echo "And this should be back on the screen"
```

Running this script directs specific output to 'myfile' while keeping other messages on the console.

2. **Setting Up Custom Input Descriptors:**
    - You can assign a new file descriptor for reading and then restore STDIN to its original source.
    - Script Demonstration:

```
#!/bin/bash
exec 6<&0
exec 0< myfile
count=1
while read line; do
  echo "Line #$count: $line"
```

```
  count=$((count + 1))
done
exec 0<&6
read -p "Are you done now? " answer
case $answer in
  y) echo "Goodbye";;
  n) echo "Sorry, this is the end.";;
esac
```

Expected Functionality: Reads from 'myfile' for the main part of the script, then restores input from the original STDIN for the final prompt.

DETAILED DISCUSSION AND APPLICATION:

1. **Understanding Descriptors**:
    - Bash allows for intricate control over where script input comes from and where output goes.
    - Custom descriptors provide flexibility for handling multiple data streams.
2. **Scripting Implications**:
    - Input redirection is particularly useful for automated data processing or when handling large datasets from files.
    - Properly managing file descriptors ensures that scripts can interact with users even after redirecting STDIN from a file.

4.8 Closing File Descriptors:

1. **Manual Closure of File Descriptors**:
    - It is often necessary to close file descriptors manually to prevent data leaks or miswrites.
    - Example:

```
#!/bin/bash
exec 3> myfile
echo "This is a test line of data" >&3
exec 3>&-
echo "This won't work" >&3
```

Expected Outcome: An error message due to attempting to write to a closed descriptor.

2. **Best Practices:**
    - Always ensure that file descriptors are closed if they are no longer needed.
    - Be mindful when reopening closed descriptors to avoid overwriting existing data.

RETRIEVING INFORMATION ABOUT OPEN HANDLES:

1. **Using lsof Command**:
    - lsof provides detailed information about all open file descriptors.
    - Example Usage:

```
lsof -a -p $$ -d 0,1,2
```

This lists details about the standard file descriptors associated with the current script.

2. **Script Example for lsof:**
    - Incorporating lsof within a script allows tracking of all used and open file descriptors.
    - Script Sample:

```
#!/bin/bash
exec 3> myfile1
exec 6> myfile2
exec 7< myfile3
lsof -a -p $$ -d 0,1,2,3,6,7
```

This script demonstrates opening additional file descriptors and then lists them using lsof.

OUTPUT SUPPRESSION TECHNIQUES:

1. **Redirecting to /dev/null**:
    - Redirecting output to /dev/null effectively silences unnecessary command output.
    - Usage Example:

```
ls -al non_existent_file 2> /dev/null
```

This suppresses error messages from the ls command.

2. **Clearing File Content:**
    - Use ***cat /dev/null > myfromnullfile*** to empty a file's contents without deleting the file itself.

3. **Append to a File using the tee Command:** ***tee*** is a Linux command-line utility that reads from the standard input and writes to **both** standard output and one or more files at the same time.
    - By default, the tee command overwrites the specified file. To append the output to the file use tee with the -a (--append) option:

```
echo "this is a new line"  | tee -a file.txtCopy
```

    - If you don't want tee to write to the standard output, redirect it to /dev/null:

```
echo "this is a new line"  | tee -a file.txt >/dev/nullCopy
```

    - The advantage of using the tee command over the >> operator is that tee allows you to simultaneously append text to multiple files and write to files owned by other users in conjunction with sudo.

- To append text to a file that you don’t have write permissions to, prepend sudo before tee as shown below:

```
echo "this is a new line" | sudo tee -a file.txtCopy
```

- tee receives the output of the echo command, elevates the sudo permissions, and writes to the file.
- To append text to more than one file, specify the files as arguments to the tee command:

```
echo "this is a new line"  | tee -a file1.txt file2.txt file3.txt
```

CONCLUSIONS:

4.1 Conclusion: Command-Line Interactivity We've uncovered the potency of command-line parameters, marking an enhancement in the interactivity of our bash scripts. The dynamic adaptation to user needs becomes effortless with this understanding.

4.2 Conclusion: Shift Command and Command-Line Switches The shift command, coupled with command-line switches, has been highlighted as instrumental tools, providing our scripts with exceptional flexibility and a structured means of interaction for users.

4.3 Conclusion: Keys versus Parameters Distinguishing between command-line keys and parameters, and the effective management of user input, are vital in bolstering script flexibility and enhancing the user experience with bash scripts.

4.4 Outcome and Conclusion: Secure Data Handling We've navigated through the essential techniques for securely handling sensitive information and the proficient processing of file data within bash scripts.

4.5 Conclusion: Input and Output Stream Management An essential groundwork has been laid on managing input and output streams in Linux and bash scripting, enabling complex data processing and precise error handling.

4.6 Conclusion and Forward Steps: Output Redirection Mastery By mastering output redirection, we have empowered our bash scripts with more control, paving the way for refined error logs and data processing enhancements.

4.7 Conclusion and Forward Steps: Input Redirection Mastery Understanding input redirection allows for the seamless incorporation of data processing from various sources, offering a new level of versatility to our bash scripts.

4.8 Conclusion and Summary: File Descriptor Management Our exploration concluded with the nuanced management of file descriptors and the strategic suppression of script outputs, sharpening the precision and efficiency of our bash scripts.

NEXT STEPS:

Comprehensive Next Steps: As we advance further in our scripting journey, let's look forward to:

- Delving into advanced options, switches, and their applications.
- Exploring complex scenarios with shift commands and structured inputs.
- Experimenting with dynamic inputs using the read command.

- Applying techniques learned in secure and efficient data handling.
- Practicing STDOUT and STDERR redirections to solidify our understanding.
- Preparing for advanced data manipulation and logging strategies with piping and tee commands.

PRACTICE EXERCISES/PROJECTS:

- **Variable Utilization Project**: Create a script that takes user input for different file names and uses command-line parameters to perform file operations.
- **Flexible Argument Processor**: Design a script using the shift command to handle an unspecified number of arguments and perform actions based on their content.
- **Key and Parameter Differentiation**: Write a script that accepts command-line options and parses them into actions, distinguishing between options (keys) and other arguments (parameters).
- **Secure Input Script**: Construct a script that prompts the user for sensitive data, handles it securely, and logs actions without exposing the data.
- **I/O Stream Handler**: Develop a script that redirects output based on user input, categorizing data into different log files.
- **File Descriptor Exercise**: Implement a bash script that opens, writes to, and closes multiple file descriptors, then use lsof to verify your script's file usage.

NEXT SEMINAR TITLE:

Management of Processes and Processors: Prepare to dive into an advanced computational landscape in our next seminar, where we will unravel the complexities of:

- **Concurrent and Parallel Processes**: Understand the principles that allow processes to run alongside or in collaboration with one another.
- **Process Execution Planning Algorithms**: Delve into the algorithms that determine the execution sequence of processes, optimizing efficiency and processor use.
- **Interaction and Collaboration of Parallel Processes**: Explore the mechanisms that enable processes to interact and work together in a parallel computing environment.
- **Process Interlocking**: Learn about techniques to avoid conflicts and ensure data integrity when multiple processes access shared resources.

***Final Notes**: As we conclude our series, remember that mastering bash scripting, especially input and output redirection, requires practice and exploration. Experiment with different scenarios to see how they affect your scripts' functionality and output. Your feedback and experiences are valuable for continuous learning and improvement. Thank you for participating in this seminar series.*

Seminar Plan: Mastering Input and Output in Bash Scripts

Objective: To provide an in-depth understanding of handling input and output in bash scripting, focusing on practical applications, redirection techniques, and file descriptor management.

Duration: Approximately 50-70 minutes.

Structure and Timing:

1. **Introduction to Bash Scripting and I/O Concepts (5 minutes)**
    - Brief overview of bash scripting importance.
    - Introduction to Input and Output concepts in Linux.
2. **Part 1: Command-Line Options and Switches (10 minutes)**
    - Discuss command-line parameters and their importance.
    - Introduce positional parameters and special characters.
    - Example and analysis of parameter usage and switches.
3. **Part 2: Redirecting Output in Scripts (10 minutes)**
    - Explore temporary and permanent redirection.
    - Demonstrate practical examples of redirection.
    - Discuss STDOUT, STDERR, and their significance in scripting.
4. **Part 3: Redirecting Input in Scripts (10 minutes)**
    - Explain the concept of STDIN and its redirection.
    - Provide examples showing the redirection of input from files.
    - Discuss reading from and processing file data within scripts.
5. **Part 4: Advanced Redirection Techniques (10 minutes)**
    - Introduce additional file descriptors and their usage.
    - Demonstrate creating, using, and closing custom file descriptors.
    - Discuss output suppression and practical uses in scripting.
6. **Practical Exercise and Q&A Session (10-15 minutes)**
    - Engage attendees with a hands-on scripting exercise incorporating lessons learned.
    - Open floor for questions, allowing attendees to seek clarification on complex topics or request additional examples.
7. **Conclusion and Next Steps (5 minutes)**
    - Summarize key takeaways from the seminar.
    - Provide resources for further learning and practice.
    - Encourage attendees to apply the knowledge to real-world scenarios and provide feedback.

Materials and Preparation:

- Provide a seminar booklet containing script examples and notes.

Post-Seminar Actions:

- TEST (asn usual 10 min /15 questions – next meet).
- Collect and review feedback to improve future seminars (from forums opened on online.ase.ro).