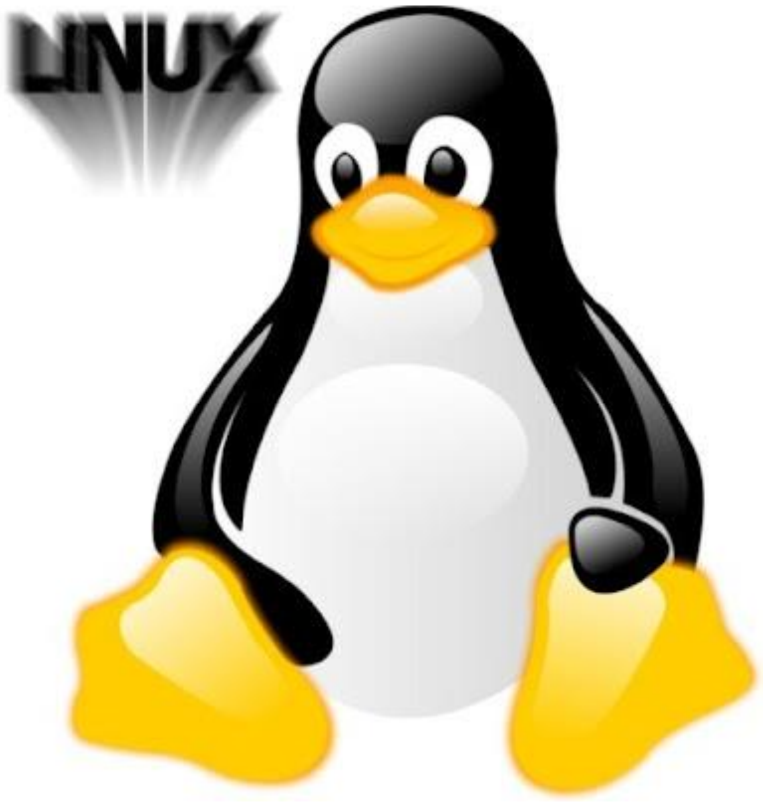


ATYPON

Instructors:
Motasim Aldiab
Fahed Jubair

done by:

Hamza Hassan

Shell Scripting Assignment

ABSTRACT

This report presents design, optimize, and benchmark a feature-rich shell developed for file analysis. And how can use Linux commands, shell scripting techniques to do this work and make some optimization strategies, and creating a user-friendly experience. It consists of three parts: implementation, optimization, and advanced feature integration.

Introduction

Shell scripting is a powerful tool for automating tasks and interacting with the operating system in a command-line environment. A shell script is a series of commands written in a scripting language that is interpreted by the shell, the command-line interface to the operating system. It allows users to execute a sequence of commands, perform complex operations, and automate repetitive tasks.
and in this report we will see how can use the shell script to generates a comprehensive report for file analysis in the given directory and its subdirectories. We will delve into various design choices, optimization techniques, advanced features, and user-friendliness aspects, evaluating their impact on performance. Additionally, we will discuss the valuable lessons we learned and the challenges we encountered during the process. Lastly, we will propose potential future improvements to enhance the script's functionality and effectiveness.

implementation and design choices:

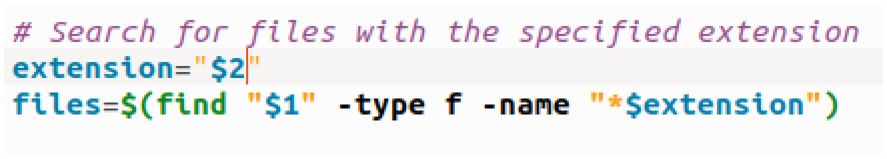
The shell script "file_analysis.sh" was developed to accomplish the following tasks:

a. Accepts a directory path as an argument.

The program accepts a directory path as an argument from the user using $1.

b. Searches for all files with a specific extension in the given directory and its subdirectories.

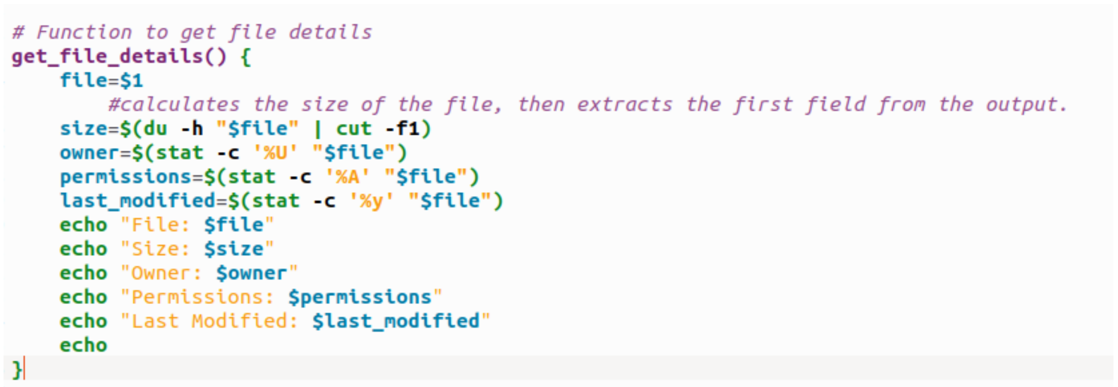
Used the "find" command to search for files with the specified extension (".txt") in the provided directory and its subdirectories



```
# Search for files with the specified extension
extension="$2"
files=$(find "$1" -type f -name "*$extension")
```

c. Generates a comprehensive report including file details such as size, owner, permissions, and last modified timestamp.

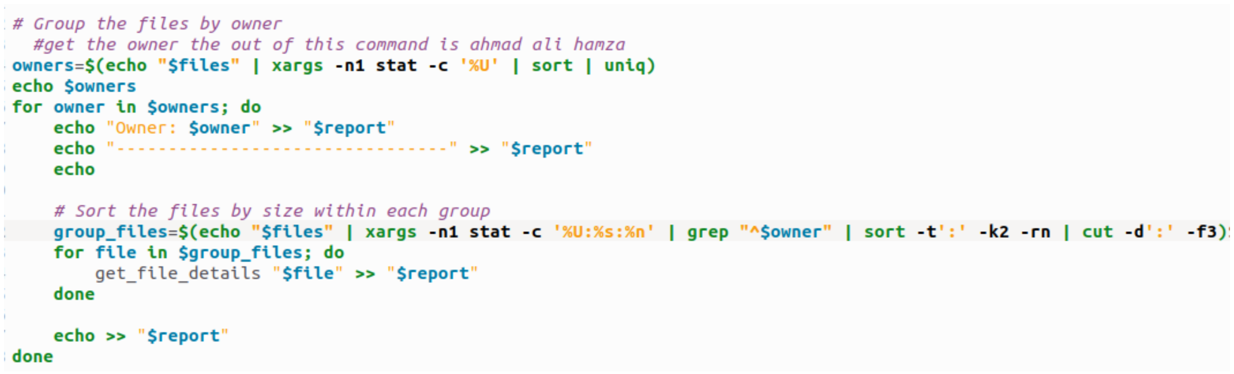
what I do is create function named “get_file_details” The function retrieves the necessary file details such as size, owner, permissions, and last modified timestamp.



```
# Function to get file details
get_file_details() {
    file=$1
        #calculates the size of the file, then extracts the first field from the output.
    size=$(du -h "$file" | cut -f1)
    owner=$(stat -c '%U' "$file")
    permissions=$(stat -c '%A' "$file")
    last_modified=$(stat -c '%y' "$file")
    echo "File: $file"
    echo "Size: $size"
    echo "Owner: $owner"
    echo "Permissions: $permissions"
    echo "Last Modified: $last_modified"
    echo
}
```

- use the "du" and "cut" commands were used to calculate the file size and extract the relevant information.
- The "stat" command was used to obtain the owner, permissions, and last modified timestamp of each file.

d. Groups the files by owner and sorts the file groups by the total size occupied by each owner.



```
# Group the files by owner
  #get the owner the out of this command is ahmad ali hamza
owners=$(echo "$files" | xargs -n1 stat -c '%U' | sort | uniq)
echo $owners
for owner in $owners; do
    echo "Owner: $owner" >> "$report"
    echo "--------------------------------" >> "$report"
    echo

    # Sort the files by size within each group
    group_files=$(echo "$files" | xargs -n1 stat -c '%U:%s:%n' | grep "^$owner" | sort -t':' -k2 -rn | cut -d':' -f3)
    for file in $group_files; do
        get_file_details "$file" >> "$report"
    done

    echo >> "$report"
done
```

1. The files were grouped by owner using the stat command and (xargs -n1 stat -c '%U') : This takes each file name provided by echo and passes it as an argument to the stat command. stat retrieves file metadata, and %U specifies that only the file owner's username should be displayed. -n1 tells xargs to process one argument at a time.
2. The owners were sorted in alphabetical order.

3. The duplicates are removed using uniq.
4. It echoes the owners variable, which will display the list of unique file owners.
5. It iterates through each owner in the owners variable.
6. For each owner:

   A. variable group_file takes the value of the files variable and passes each file as an argument to the stat command using xargs -n1.

   a. the stat command with the -c option is used to specify the format of the output.
   b. %U is used to retrieve the owner of the file.
   c. %s is used to retrieve the size of the file in bytes.
   d. %n is used to retrieve the file name.
   e. The output of stat is in the format: <owner>:<size>:<filename>

   B. The output of the stat command is piped (|) to grep to filter the lines that start with the current value of the owner variable.

   C. The filtered output is piped to sort with the -t option to specify the field separator as ':' and the -k option to specify the field to sort on (the size field -k2 in descending order -rn).

   D. then the sorted output is piped to cut with the -d option set to ':' and the -f option set to 3.

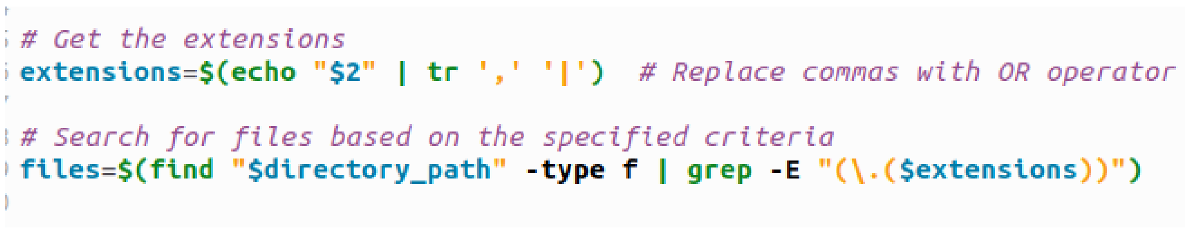
   E. executes the get_file_details function on each file belonging to the current owner. To get the details for each file

7. Save everything in $report variable it's provide text file named file_analysis ($report="file_analysis.txt".)
8. The process continues for the next owner, and so on.

Advanced Feature Integration

In this part of report, I will Enhance shell script with additional advanced features:

a. <u>Implement support for multiple file extensions, allowing users to search for files with various extensions simultaneously</u>.



```
# Get the extensions
extensions=$(echo "$2" | tr ',' '|')  # Replace commas with OR operator

# Search for files based on the specified criteria
files=$(find "$directory_path" -type f | grep -E "(\.($extensions))")
```

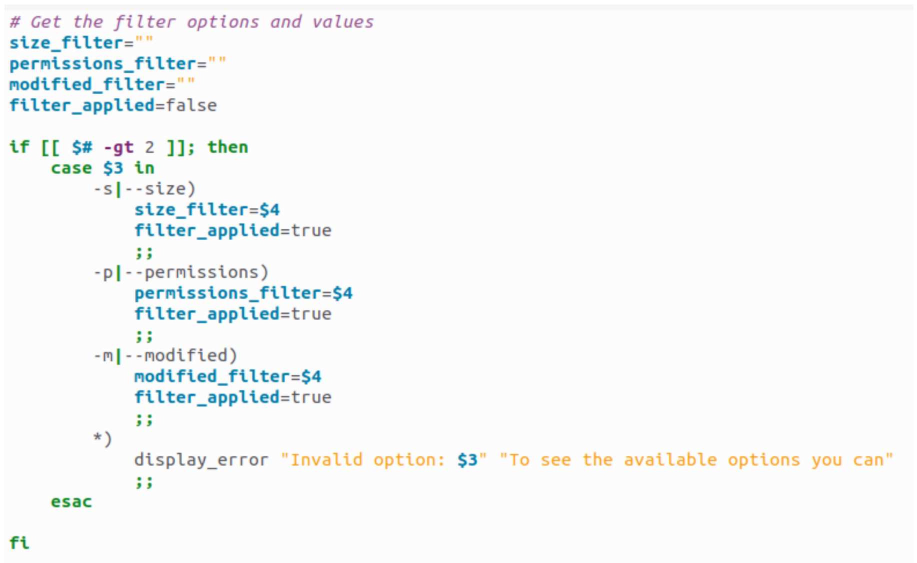
The script accepts the extensions as a parameter, provided by the user. These extensions are separated by commas. To enable searching for multiple extensions, the script utilizes the 'tr' command to replace the commas with the logical OR (|) operator. This transforms the input into a regular expression pattern that can match files with any of the specified extensions.

Searching for Files:
The find command is used to search for files and the -type f option ensures that only regular files are considered for the search.

Filtering Based on Extensions:
The output of the find command is piped to grep, which applies the extended regular expression (-E) mode. The regular expression pattern (\.($extensions)) is used to match files with extensions specified by the user. This pattern ensures that the file extension matches any of the provided extensions.

the file search is stored in the files variable, which contains the paths of the matched files.

b. <u>Include an option to filter files based on size, permissions, or last modified timestamp, allowing users to customize their search criteria</u>.



```
# Get the filter options and values
size_filter=""
permissions_filter=""
modified_filter=""
filter_applied=false

if [[ $# -gt 2 ]]; then
    case $3 in
        -s|--size)
            size_filter=$4
            filter_applied=true
            ;;
        -p|--permissions)
            permissions_filter=$4
            filter_applied=true
            ;;
        -m|--modified)
            modified_filter=$4
            filter_applied=true
            ;;
        *)
            display_error "Invalid option: $3" "To see the available options you can"
            ;;
    esac

fi
```

the implemented functionality:

Retrieving Filter Options and Values:

- The script initializes variables size_filter, permissions_filter, and modified_filter to store the provided filter values. A variable filter_applied is set to false initially.

- If the number of arguments ($#) is greater than 2, the script enters a case statement to check the third argument ($3) for available options.

Handling Filter Options:

- Size Filter (-s or --size):
  If the option -s or --size is provided, the value is assigned to the size_filter variable, and filter_applied is set to true.

- Permissions Filter (-p or --permissions):
  If the option -p or --permissions is provided, the value is assigned to the permissions_filter variable, and filter_applied is set to true.

- Modified Filter (-m or --modified):
  If the option -m or --modified is provided, the value is assigned to the modified_filter variable, and filter_applied is set to true.

- Invalid Option:
  If an invalid option is provided, an error message is displayed.

Apply filter:

```
# Apply filters if provided
if [[ $filter_applied = true ]]; then

    # Filter by size
    if [[ -n $size_filter ]]; then
        files=$(du -b $files | awk -v size="$size_filter" '$1 >= size { print $2 }')
    fi

    # Filter by permissions
    if [[ -n $permissions_filter ]]; then
        files=$(ls -l | grep "$permissions_filter" | awk '{ print $NF }')

    fi

    # Filter by last modified timestamp
    if [[ -n $modified_filter ]]; then
        files=$(find $files -type f -mtime -$modified_filter)
    fi
fi
```

If filter_applied is true, the script applies the specified filters.

Size Filter:
If a size filter value is provided, the du command is used to retrieve the file sizes (-b option bytes) of the files and the awk command filters the files based on the provided size.

Permissions Filter:

- ls -l: The ls command lists files in the current directory with detailed information, and the -l option displays the output in long format, including permissions.

- grep "$permissions_filter": The output of the ls -l command is piped (|) to the grep command. The grep command filters the lines that match the specified permissions filter ($permissions_filter). The variable $permissions_filter contains the filter value provided by the user.

- awk '{ print $NF }': The output of the grep command is piped (|) to the awk command. The awk command processes the input and prints only the last field ($NF) from each line. In this case, it extracts and outputs the file names that match the permissions filter.

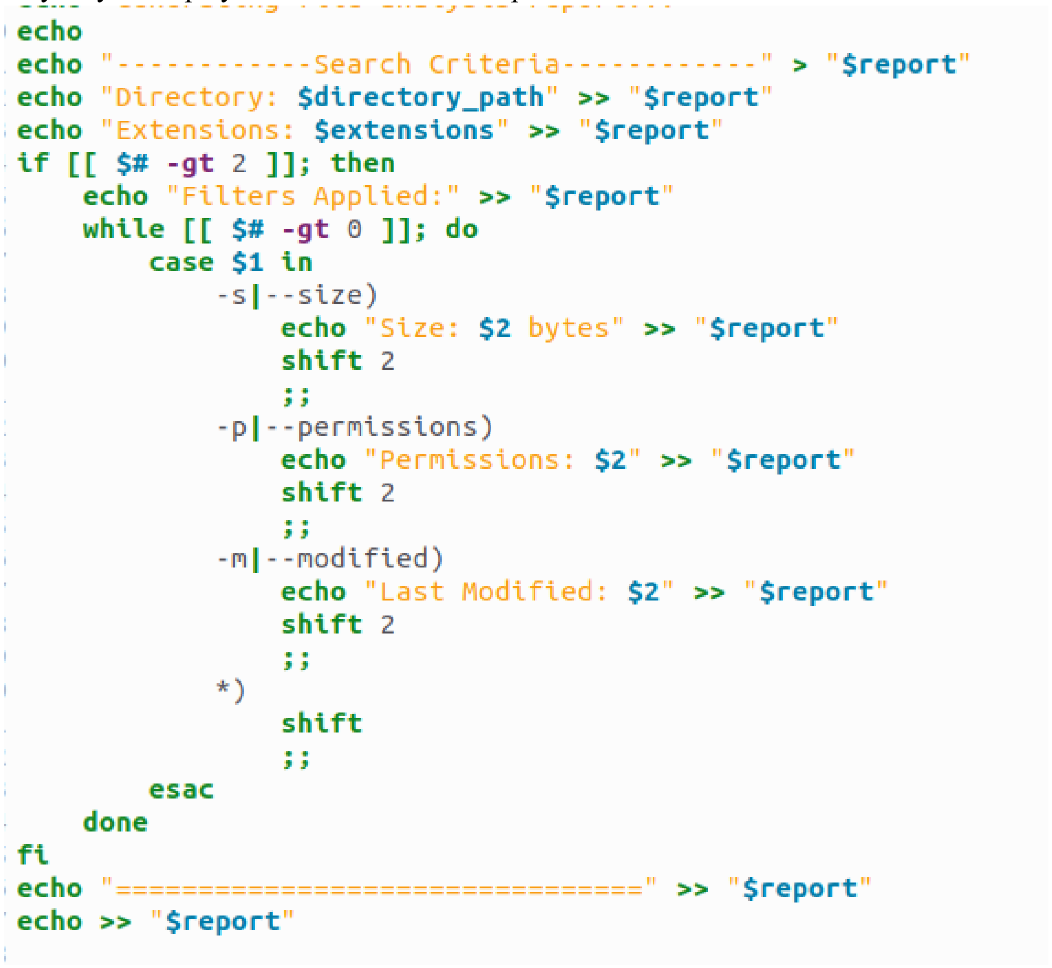
Modified Filter:

-mtime -$modified_filter: This option filters files based on their modification time. The value $modified_filter is a variable representing the number of days. It is used to filter files based on their modification time relative to the current date.

The (-) before $modified_filter indicates a negative value, meaning it searches for files modified "less than" (-) the specified number of days ago. This will include files that were modified within the last $modified_filter days.

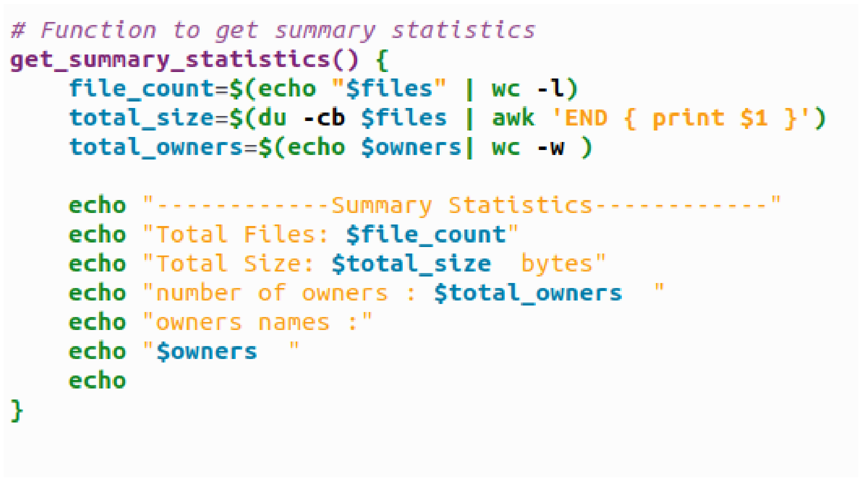
For example, if $modified_filter is set to 7, the command will find files modified within the last 7 days.

they way of display the search criteria in report :



```
echo
echo "------------Search Criteria------------" > "$report"
echo "Directory: $directory_path" >> "$report"
echo "Extensions: $extensions" >> "$report"
if [[ $# -gt 2 ]]; then
    echo "Filters Applied:" >> "$report"
    while [[ $# -gt 0 ]]; do
        case $1 in
            -s|--size)
                echo "Size: $2 bytes" >> "$report"
                shift 2
                ;;
            -p|--permissions)
                echo "Permissions: $2" >> "$report"
                shift 2
                ;;
            -m|--modified)
                echo "Last Modified: $2" >> "$report"
                shift 2
                ;;
            *)
                shift
                ;;
        esac
    done
fi
echo "================================" >> "$report"
echo >> "$report"
```

c. <u>Enable the script to generate a summary report that displays total file count, total size, and other relevant statistics</u>



```
# Function to get summary statistics
get_summary_statistics() {
    file_count=$(echo "$files" | wc -l)
    total_size=$(du -cb $files | awk 'END { print $1 }')
    total_owners=$(echo $owners| wc -w )

    echo "------------Summary Statistics------------"
    echo "Total Files: $file_count"
    echo "Total Size: $total_size  bytes"
    echo "number of owners : $total_owners   "
    echo "owners names :"
    echo "$owners   "
    echo
}
```

- file_count=$(echo "$files" | wc -l): This line counts the number of lines in the $files variable. Since the $files variable contains a list of file paths (each path on a separate line), this command counts the number of files in the list and assigns the result to the file_count variable.

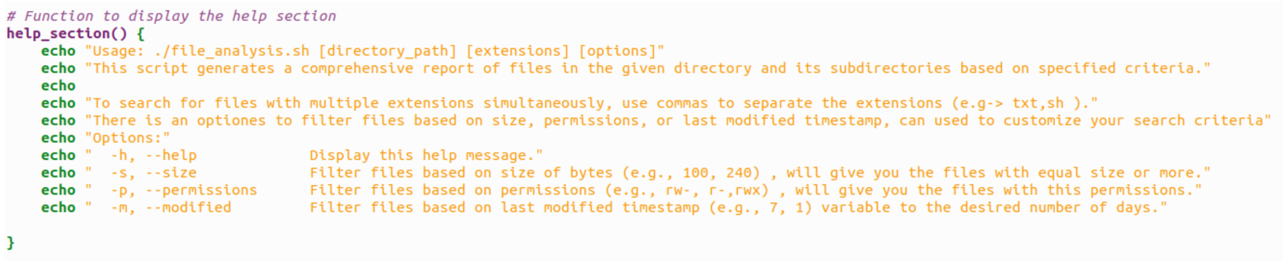
- total_size=$(du -cb $files | awk 'END { print $1 }'): This line calculates the total size of the files in the $files list. It uses the du command with the -cb options to print the cumulative size of all files (-c produce a grand total). The output is then piped to awk, which extracts the first column of the last line (corresponding to the total size) and assigns it to the total_size variable.

- total_owners=$(echo $owners| wc -w): This line counts the number of words (owners) in the $owners variable. It uses the wc command with the -w option to count the number of words and assigns the result to the total_owners variable.

User-friendliness aspects

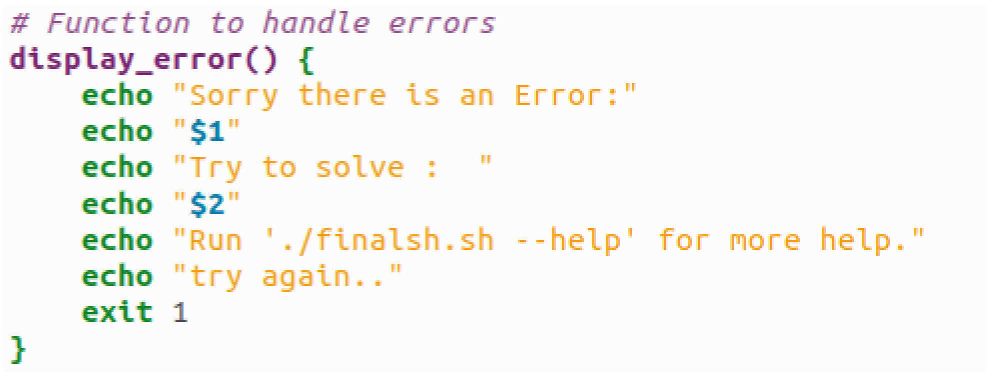
Here are some user-friendliness aspects in the code:

1. Help Section: The script provides a help section that explains the usage of the script, including command-line arguments and options. Users can run the script with the "--help" or "-h" option to get detailed information about how to use the script.



```
# Function to display the help section
help_section() {
    echo "Usage: ./file_analysis.sh [directory_path] [extensions] [options]"
    echo "This script generates a comprehensive report of files in the given directory and its subdirectories based on specified criteria."
    echo
    echo "To search for files with multiple extensions simultaneously, use commas to separate the extensions (e.g-> txt,sh )."
    echo "There is an optiones to filter files based on size, permissions, or last modified timestamp, can used to customize your search criteria"
    echo "Options:"
    echo "  -h, --help            Display this help message."
    echo "  -s, --size            Filter files based on size of bytes (e.g., 100, 240) , will give you the files with equal size or more."
    echo "  -p, --permissions     Filter files based on permissions (e.g., rw-, r-,rwx) , will give you the files with this permissions."
    echo "  -m, --modified        Filter files based on last modified timestamp (e.g., 7, 1) variable to the desired number of days."

}
```

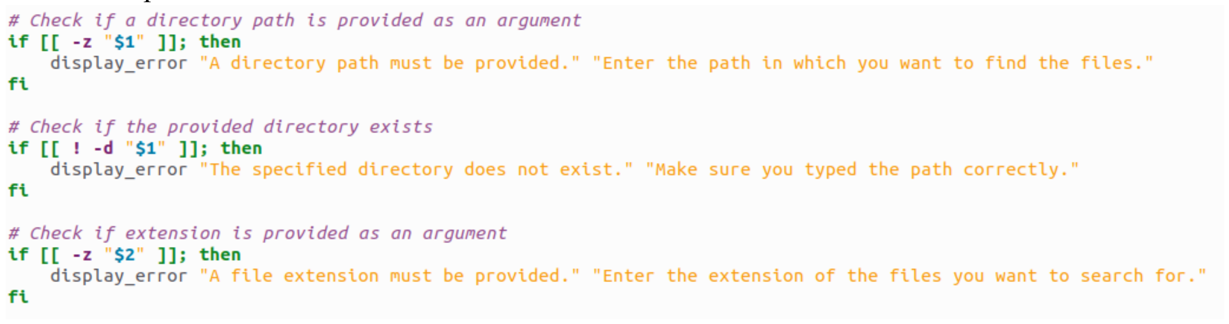
2. Error Handling: The script includes error handling to provide meaningful error messages in case of incorrect usage or invalid input. It displays specific error messages indicating the cause of the error and suggests possible solutions to the user.



```
# Function to handle errors
display_error() {
    echo "Sorry there is an Error:"
    echo "$1"
    echo "Try to solve :  "
    echo "$2"
    echo "Run './finalsh.sh --help' for more help."
    echo "try again.."
    exit 1
}
```

The display_error function is responsible for displaying error messages. It takes two arguments: the error message and the suggestion for resolution. It formats and displays the error message along with the provided suggestions. This helps users understand what went wrong and provides guidance on how to resolve the issue.

it's used in these cases:
Missing Directory Path, Invalid Directory, Missing File Extension, Invalid Option, File Not Found.
Some example :

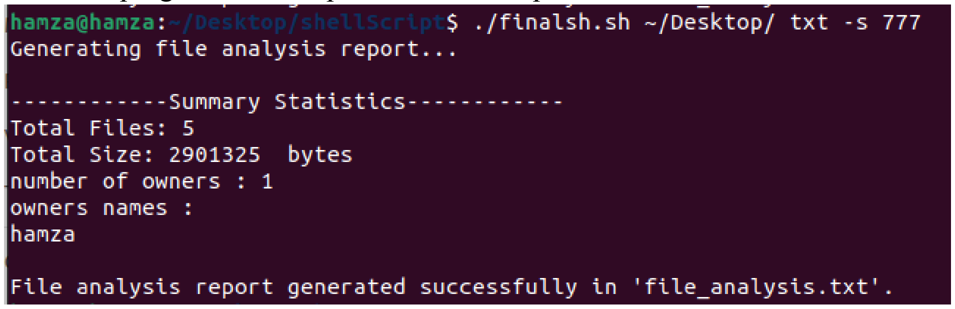


```
# Check if a directory path is provided as an argument
if [[ -z "$1" ]]; then
    display_error "A directory path must be provided." "Enter the path in which you want to find the files."
fi

# Check if the provided directory exists
if [[ ! -d "$1" ]]; then
    display_error "The specified directory does not exist." "Make sure you typed the path correctly."
fi

# Check if extension is provided as an argument
if [[ -z "$2" ]]; then
    display_error "A file extension must be provided." "Enter the extension of the files you want to search for."
fi
```

3. The script generates a report in a structured manner, making it easy for users to read and understand the information. The report includes sections such as search criteria, file details, and summary statistics, making it comprehensive and organized.

4. The script allows users to apply filters based on file size, permissions, and last modified timestamp. This flexibility enables users to customize their search criteria according to their requirements.

5. The script provides progress updates by displaying messages during the execution, such as "Generating file analysis report..." and "File analysis report generated successfully." These messages inform the user about the progress and completion of the script's task.



```
hamza@hamza:~/Desktop/shellScript$ ./finalsh.sh ~/Desktop/ txt -s 777
Generating file analysis report...

------------Summary Statistics------------
Total Files: 5
Total Size: 2901325  bytes
number of owners : 1
owners names :
hamza

File analysis report generated successfully in 'file_analysis.txt'.
```

Lessons learned

1. Script Documentation: Including a help section in the script that explains the usage, options, and parameters is essential. It helps users understand how to use the script and provides guidance when they need assistance.

2. Error Handling: Implementing error handling is crucial to provide meaningful error messages when something goes wrong. It helps users understand the issue and suggests possible solutions.

3. Modularity: Dividing the code into functions improves code organization and reusability. Each function can handle a specific task, making the code easier to read, understand, and maintain.

4. Command-Line Arguments: The script parses command-line arguments using conditional statements and handles different options and their corresponding values. Learning how to handle command-line arguments is crucial for building flexible and customizable scripts.

5. Text Processing: The script uses various text processing commands, such as cut, awk, grep, xargs, and sort, to manipulate and extract relevant information from text output. Familiarity with these commands and their usage is valuable for working with text data in shell scripting.

6. Output Formatting: The script generates a formatted report in a text file, including headers, separators, and organized information. Knowing how to format and present data in a structured manner enhances the usability and readability of the generated output.

7. Collaboration and Reusability: Sharing the script with others for collaboration or reuse requires clear documentation, standardized coding practices, and ensuring that the script works correctly in different environments. It's essential to write code that can be easily understood, modified, and shared.

8. About owner: how can easily create new owner in linux using useradd command and change the owner of the file using chown command .

9. About group: how can easily create new group in linux using groupadd command and change the group of the file using chown command .

10. Testing and Debugging: Before finalizing the script, thorough testing and debugging should be performed to ensure its correctness and handle possible edge cases. Proper testing helps identify and fix issues early in the development process.

Difficulties

Some difficulties in debugging the code, handling errors that may occur, and covering all possible errors, such as trying more than one owner and more than one group

Potential future improvements

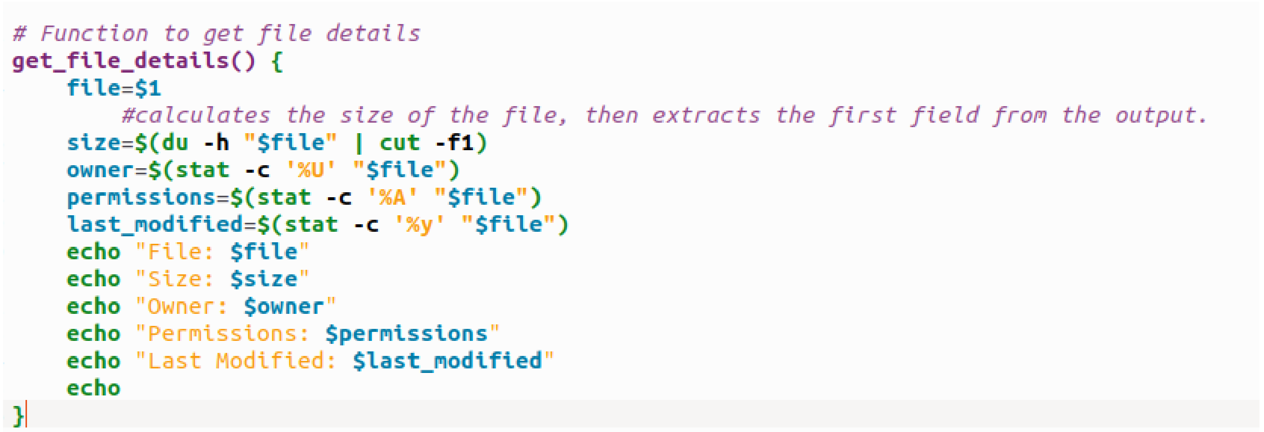
1. Extend the script to support additional filtering options, such as filtering by file creation timestamp, file type, or file contents. This would provide more flexibility to users when searching for specific files.

2. Implement the ability to exclude certain files or directories from the search results. This would be useful when users want to exclude specific files or directories from the analysis.

3. Allow users to customize the output format of the generated report. For example, provide options to generate the report in different formats (e.g., JSON, CSV) or allow users to choose specific fields to include or exclude from the report.

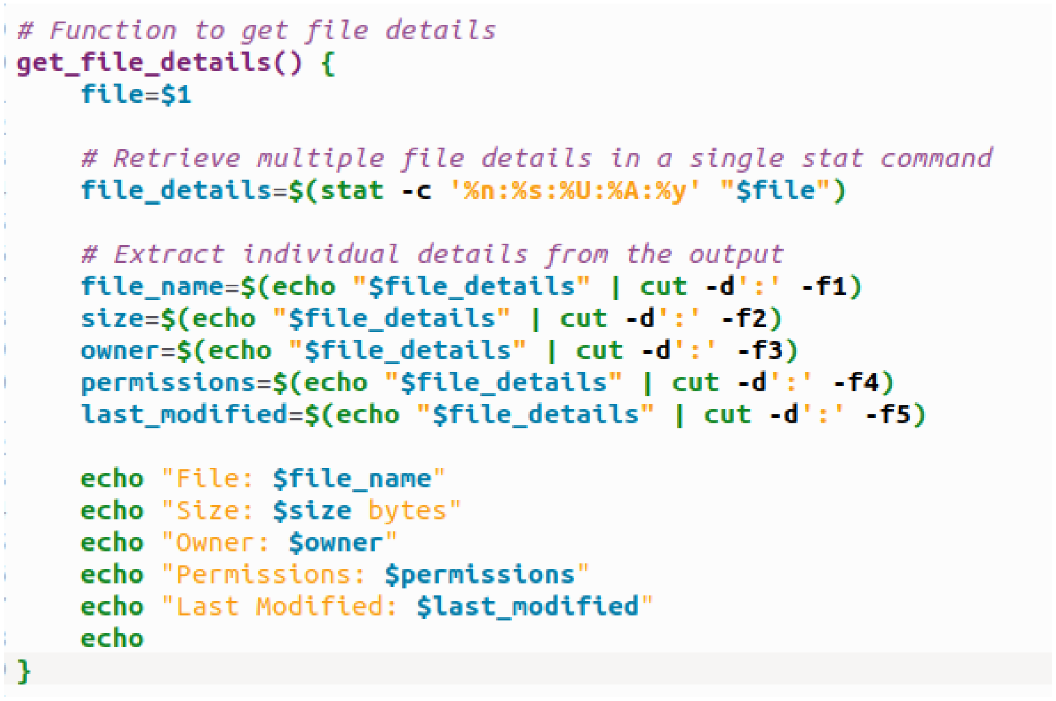
optimization techniques

1. Optimized File Details Retrieval: The get_file_details function retrieves important file details (size, owner, permissions, last modified) using the stat command, avoiding the need for multiple system calls for each detail. This optimization improves performance when analyzing a large number of files.



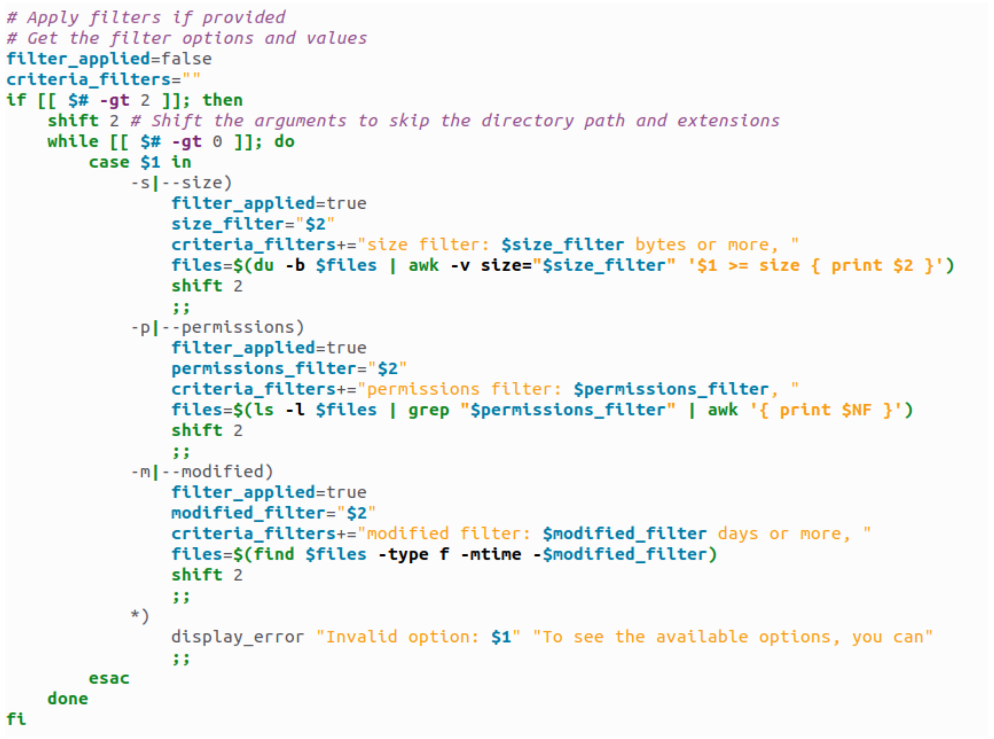
```
# Function to get file details
get_file_details() {
    file=$1
        #calculates the size of the file, then extracts the first field from the output.
    size=$(du -h "$file" | cut -f1)
    owner=$(stat -c '%U' "$file")
    permissions=$(stat -c '%A' "$file")
    last_modified=$(stat -c '%y' "$file")
    echo "File: $file"
    echo "Size: $size"
    echo "Owner: $owner"
    echo "Permissions: $permissions"
    echo "Last Modified: $last_modified"
    echo
}
```

2. Reduce System Calls: Minimize the number of system calls by retrieving multiple file details in a single stat command. Instead of calling stat multiple times for each file detail (size, owner, permissions, last modified), you can modify the get_file_details function to retrieve all the required details in a single stat command. This will reduce the overhead of multiple system calls.



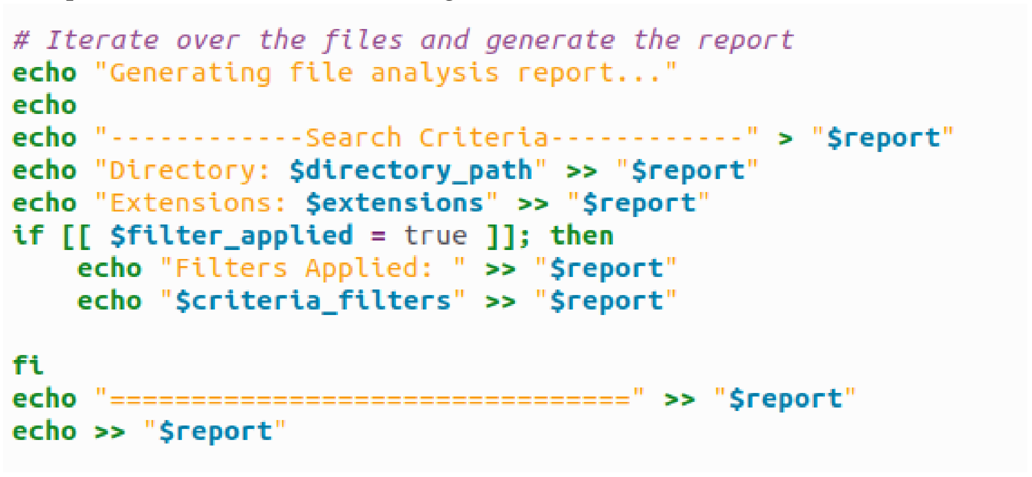
```
# Function to get file details
get_file_details() {
    file=$1

    # Retrieve multiple file details in a single stat command
    file_details=$(stat -c '%n:%s:%U:%A:%y' "$file")

    # Extract individual details from the output
    file_name=$(echo "$file_details" | cut -d':' -f1)
    size=$(echo "$file_details" | cut -d':' -f2)
    owner=$(echo "$file_details" | cut -d':' -f3)
    permissions=$(echo "$file_details" | cut -d':' -f4)
    last_modified=$(echo "$file_details" | cut -d':' -f5)

    echo "File: $file_name"
    echo "Size: $size bytes"
    echo "Owner: $owner"
    echo "Permissions: $permissions"
    echo "Last Modified: $last_modified"
    echo
}
```

3. Multiple Filter Application: optimize the code to be able to take more than one file filter at one time and this will give the user greater ability to customize the files he wants improved efficiency



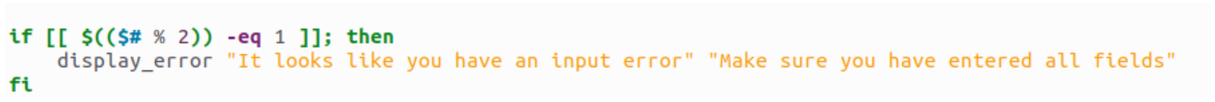
```
# Apply filters if provided
# Get the filter options and values
filter_applied=false
criteria_filters=""
if [[ $# -gt 2 ]]; then
    shift 2 # Shift the arguments to skip the directory path and extensions
    while [[ $# -gt 0 ]]; do
        case $1 in
            -s|--size)
                filter_applied=true
                size_filter="$2"
                criteria_filters+="size filter: $size_filter bytes or more, "
                files=$(du -b $files | awk -v size="$size_filter" '$1 >= size { print $2 }')
                shift 2
                ;;
            -p|--permissions)
                filter_applied=true
                permissions_filter="$2"
                criteria_filters+="permissions filter: $permissions_filter, "
                files=$(ls -l $files | grep "$permissions_filter" | awk '{ print $NF }')
                shift 2
                ;;
            -m|--modified)
                filter_applied=true
                modified_filter="$2"
                criteria_filters+="modified filter: $modified_filter days or more, "
                files=$(find $files -type f -mtime -$modified_filter)
                shift 2
                ;;
            *)
                display_error "Invalid option: $1" "To see the available options, you can"
                ;;
        esac
    done
fi
```

shift 2: This action skipped the directory path and extensions, allowing the loop to focus solely on the filter options. Improve the way of displaying the search criteria on report using the variable criteria_filter and this will improve code performance Because we no longer need to use if statements like before



```
# Iterate over the files and generate the report
echo "Generating file analysis report..."
echo
echo "-----------Search Criteria------------" > "$report"
echo "Directory: $directory_path" >> "$report"
echo "Extensions: $extensions" >> "$report"
if [[ $filter_applied = true ]]; then
    echo "Filters Applied: " >> "$report"
    echo "$criteria_filters" >> "$report"

fi
echo "=================================" >> "$report"
echo >> "$report"
```

Add a new error handler to make sure the argument count is correct or something is missing



```
if [[ $(($# % 2)) -eq 1 ]]; then
    display_error "It looks like you have an input error" "Make sure you have entered all fields"
fi
```

4. Add suggestions for resolution to the error message.

Conclusion

In conclusion, this report has presented the design choices, optimization techniques, advanced features, user-friendliness aspects, and their impact on performance. Include a reflection section discussing the lessons learned, difficulties encountered, and potential future improvements

In conclusion, this report has presented shell script developed for file analysis. Through the use of Linux commands, shell scripting techniques, and has presented the design choices, optimization techniques, advanced features, user-friendliness aspects, and their impact on performance, the lessons learned, difficulties , potential future improvements and the script successfully accomplishes various tasks related to file analysis.

<u>The end</u>