



Bash Scripting

Mensi Mohamed Amine

Abstract

This script explores Bash, a Unix shell and command language, focusing on its automation capabilities for routine tasks in Linux systems.

1 Introduction

Bash scripting simplifies complex workflows by combining commands into executable scripts, making it a vital tool for system administrators and developers.

Contents

1	Introduction	1
2	Bash Scripting Concepts	7
2.1	Basic Concepts	7
2.2	Special Variables	7
2.3	Input/Output	7
2.4	Conditionals	7
2.5	Loops	8
2.6	Arrays	8
2.7	Functions	9
2.8	String Manipulation	9
2.9	Arithmetic	9
2.10	File Operations	9
2.11	Process Control	10
2.12	Advanced Features	10
2.13	Best Practices	10
3	Basic Concepts in Bash Scripting	11
3.1	Script Structure	11
3.2	Variables	11
3.3	User Input	11
3.4	Command Substitution	11
3.5	Arithmetic Operations	11
3.6	Basic Conditionals	12
3.7	Loops	12
3.8	Exit Status	12
3.9	Basic File Operations	12
3.10	Functions	13
4	Special Variables in Bash Scripting	14
4.1	Positional Parameters	14
4.2	Script Status Variables	14
4.3	Special Shell Variables	14
4.4	Example Usage	14
4.5	Key Differences Between \$* and \$@	15
4.6	Best Practices	15
4.7	Common Exit Status Codes	15
5	Input/Output in Bash Scripting	16
5.1	Standard Output (stdout)	16
5.2	Standard Input (stdin)	16
5.3	File Operations	16
5.4	Redirection	16
5.5	Here Documents & Strings	17
5.6	Special File Descriptors	17
5.7	Practical Examples	17
5.8	Best Practices	18

6 Conditionals in Bash Scripting	19
6.1 Basic if-then-else Structure	19
6.2 Test Conditions (Square Brackets [])	19
6.3 Advanced Conditional Expressions ([[]])	19
6.4 Logical Operators	20
6.5 Case Statements	20
6.6 Arithmetic Conditions	20
6.7 Combining Conditions	20
6.8 Common Pitfalls	21
6.9 Practical Examples	21
7 Loops in Bash Scripting	22
7.1 For Loops	22
7.2 While Loops	22
7.3 Until Loops	23
7.4 Loop Control Statements	23
7.5 Practical Examples	23
7.6 Best Practices	24
7.7 Nested Loops Example	24
7.8 Special Loop Forms	24
8 Arrays in Bash Scripting	25
8.1 Array Basics	25
8.2 Working with Indexed Arrays	25
8.3 Working with Associative Arrays (Bash 4+)	25
8.4 Iterating Through Arrays	26
8.5 Array Manipulation	26
8.6 Practical Examples	27
8.7 Best Practices	27
8.8 Common Pitfalls	27
9 Functions in Bash Scripting	28
9.1 Basic Function Syntax	28
9.2 Function Parameters	28
9.3 Return Values	28
9.4 Variable Scope	29
9.5 Advanced Function Features	29
9.6 Practical Examples	30
9.7 Best Practices	30
9.8 Common Pitfalls	31
10 String Manipulation in Bash Scripting	32
10.1 Basic String Operations	32
10.2 Substring Operations	32
10.3 Pattern Matching and Replacement	32
10.4 String Splitting and Joining	33
10.5 String Validation and Testing	33
10.6 Advanced String Processing	33
10.7 Practical Examples	34
10.8 Best Practices	34

11 Arithmetic in Bash Scripting	36
11.1 Basic Arithmetic Methods	36
11.1.1 Double Parentheses ((()))	36
11.1.2 Dollar Double Parentheses \${(())}	36
11.1.3 let Command	36
11.1.4 expr Command (legacy)	36
11.2 Arithmetic Operators	36
11.2.1 Basic Operators	36
11.2.2 Bitwise Operators	36
11.2.3 Assignment Operators	37
11.3 Number Bases	37
11.3.1 Different Base Representation	37
11.3.2 Base Conversion	37
11.4 Floating-Point Arithmetic	37
11.4.1 Using bc (for floating-point)	37
11.4.2 Using awk	37
11.5 Increment/Decrement Operations	37
11.5.1 Postfix and Prefix	37
11.6 Practical Examples	38
11.6.1 Simple Calculator	38
11.6.2 Temperature Conversion	38
11.6.3 Prime Number Check	38
11.7 Best Practices	38
11.8 Common Pitfalls	38
12 File Operations in Bash Scripting	40
12.1 File Testing and Inspection	40
12.1.1 Test Operators	40
12.1.2 Get File Information	40
12.2 File Reading and Writing	40
12.2.1 Read Entire File	40
12.2.2 Read Line by Line	40
12.2.3 Write to File	40
12.2.4 Here Documents	40
12.3 File Manipulation	41
12.3.1 Copy, Move, Rename	41
12.3.2 Create and Remove	41
12.3.3 Find Files	41
12.4 File Permissions	41
12.4.1 Change Permissions	41
12.4.2 Change Ownership	41
12.5 Advanced Operations	41
12.5.1 Temporary Files	41
12.5.2 File Locking	41
12.5.3 Compare Files	41
12.6 Practical Examples	42
12.6.1 Backup Script	42
12.6.2 Log Rotator	42
12.6.3 File Search and Process	42
12.7 Best Practices	42
12.8 Common Pitfalls	43

13 Process Control in Bash Scripting	44
13.1 Process Basics	44
13.1.1 Running Processes	44
13.1.2 Process Status	44
13.2 Job Control	44
13.2.1 Managing Background Jobs	44
13.2.2 Job Notification	44
13.3 Process Termination	44
13.3.1 Killing Processes	44
13.3.2 Graceful Shutdown	44
13.4 Process Monitoring	45
13.4.1 Watch Processes	45
13.4.2 Timeout Control	45
13.5 Advanced Process Management	45
13.5.1 Process Substitution	45
13.5.2 Named Pipes	45
13.5.3 Process Priority	45
13.6 Practical Examples	45
13.6.1 Process Monitor Script	45
13.6.2 Parallel Processing	46
13.6.3 Service Manager	46
13.7 Best Practices	46
13.8 Common Pitfalls	46
14 Advanced Features in Bash Scripting	47
14.1 Arrays and Associative Arrays	47
14.1.1 Indexed Arrays	47
14.1.2 Associative Arrays (Bash 4+)	47
14.2 Process Substitution	47
14.2.1 Compare outputs without temp files	47
14.3 Coprocesses	47
14.3.1 Two-way communication with processes	47
14.4 Named Pipes (FIFOs)	47
14.4.1 Create persistent pipes	47
14.5 Indirect References	47
14.5.1 Dynamic variable names	47
14.6 Parameter Transformation	48
14.6.1 Advanced string manipulation	48
14.7 Regular Expressions	48
14.7.1 Native regex support (Bash 3.2+)	48
14.8 Advanced I/O Redirection	48
14.8.1 Custom file descriptors	48
14.9 Error Handling	48
14.9.1 Advanced trap usage	48
14.10 Namespace Control	48
14.10.1 Restricted shells	48
14.10.2 Function namespaces	48
14.11 Practical Examples	49
14.11.1 Dynamic Function Dispatch	49
14.11.2 Concurrent Process Manager	49
14.11.3 Self-Modifying Script	49
14.12 Best Practices	50

15 Best Practices in Bash Scripting	51
15.1 Script Structure and Organization	51
15.1.1 Header Section	51
15.1.2 Configuration Section	51
15.1.3 Function Definitions	51
15.1.4 Main Execution	51
15.2 Error Handling	51
15.2.1 Strict Mode	51
15.2.2 Custom Error Handling	51
15.2.3 Exit Codes	52
15.3 Security Practices	52
15.3.1 Input Validation	52
15.3.2 Sanitization	52
15.3.3 Privilege Management	52
15.4 Coding Standards	52
15.4.1 Variable Naming	52
15.4.2 Function Practices	52
15.4.3 Commenting	52
15.5 Performance Considerations	53
15.5.1 Subshell Minimization	53
15.5.2 Builtin Preference	53
15.6 Portability	53
15.6.1 Shebang Portability	53
15.6.2 Feature Detection	53
15.6.3 Cross-Platform Paths	53
15.7 Documentation	53
15.7.1 Help Function	53
15.7.2 Inline Documentation	53
15.8 Testing and Debugging	54
15.8.1 Debug Mode	54
15.8.2 Linting	54
15.8.3 Unit Testing	54
15.9 Maintenance Practices	54
15.9.1 Version Control	54
15.9.2 Logging	54
15.9.3 Dependency Management	54
15.10 Practical Examples	54
15.10.1 Robust Template	54

2 Bash Scripting Concepts

2.1 Basic Concepts

- **Shebang:** `#!/bin/bash` – specifies the interpreter
- **Variables:**

```
var="value"
echo $var
```

- **Quoting:**
 - Single quotes (`'`) – literal
 - Double quotes (`"`) – allows variable expansion
 - Backticks (`'`) – command substitution (deprecated, use `$()` instead)

2.2 Special Variables

- `$0` – script name
- `$1-$9` – positional parameters
- `$#` – number of arguments
- `$` – all arguments as single string
- `$` – all arguments as separate strings
- `$?` – exit status of last command
- `$$` – process ID
- `$!` – process ID of last background command

2.3 Input/Output

Output:

```
echo "Hello"
printf "Hello %s\n" $name
```

Input:

```
read variable
read -p "Prompt: " variable
```

2.4 Conditionals

If-else:

```
if [ condition ]; then
  commands
elif [ condition ]; then
  commands
else
  commands
fi
```

Test conditions ([] or test):

- File tests: -f, -d, -e
- String tests: =, !=, -z, -n
- Numeric tests: -eq, -ne, -lt, -le, -gt, -ge

Case statement:

```
case $var in
  pattern1) commands ;;
  pattern2) commands ;;
  *) default commands ;;
esac
```

2.5 Loops

For loop:

```
for var in list; do
  commands
done
```

C-style for loop:

```
for ((i=0; i<10; i++)); do
  commands
done
```

While loop:

```
while [ condition ]; do
  commands
done
```

Until loop:

```
until [ condition ]; do
  commands
done
```

2.6 Arrays

Indexed arrays:

```
arr=("one" "two" "three")
echo ${arr[1]} # two
```

Associative arrays (Bash 4+):

```
declare -A arr
arr["key"]="value"
```

2.7 Functions

Definition:

```
function_name() {
    commands
    return value
}
```

- Parameters: \$1, \$2, etc.
- Return values: use `return` for status, capture output for data

2.8 String Manipulation

- Substring:

```
 ${string:position:length}
```

- Replacement:

```
 ${string/substring/replacement}
 ${string//substring/replacement}
```

- Length:

```
 ${#string}
```

2.9 Arithmetic

Integer arithmetic:

```
$((expression))
```

let command:

```
let "var = 5 + 3"
```

2.10 File Operations

Reading files:

```
while IFS= read -r line; do
    echo "$line"
done < file.txt
```

Redirection:

- > – overwrite
- >> – append
- 2> – stderr
- &> – stdout and stderr

2.11 Process Control

Background jobs:

```
command &
```

Job control:

- `jobs` – list background jobs
- `fg` – bring to foreground
- `bg` – continue in background
- `kill` – terminate process

2.12 Advanced Features

Traps (signal handling):

```
trap "commands" SIGNAL
```

Debugging:

- `set -x`, `set +x`
- `bash -x script.sh`

Here documents:

```
command << EOF
text
EOF
```

Here strings:

```
command <<< "string"
```

Command substitution:

```
var=$(command)
```

Process substitution:

```
diff <(command1) <(command2)
```

2.13 Best Practices

- Always quote variables
- Use `[[]]` for conditional tests
- Check for required commands at start
- Validate input
- Use `set -e` to exit on error
- Use `set -u` to fail on unset variables
- Include help/usage information

3 Basic Concepts in Bash Scripting

Here are the fundamental concepts you need to understand to start writing Bash scripts:

3.1 Script Structure

- **Shebang:** First line (`#!/bin/bash`) tells the system this is a Bash script
- **Comments:** Lines starting with # (except shebang)
- **Commands:** Written one per line or separated by semicolons

```
#!/bin/bash
# This is a comment
echo "Hello World"
```

3.2 Variables

- **Declaration:** `name="value"` (no spaces around =)
- **Usage:** `$name` or `${name}`
- **Types:** All variables are strings by default

```
name="Alice"
echo "Hello $name"    # Hello Alice
echo "Hello ${name}"  # Hello Alice
```

3.3 User Input

- **Reading input:** `read` command
- **Prompting:** `read -p "Prompt" variable`

```
read -p "Enter your name: " username
echo "Welcome, $username"
```

3.4 Command Substitution

- Execute a command and use its output
- Syntax: `$(command)` or backticks (deprecated)

```
current_date=$(date)
echo "Today is $current_date"
```

3.5 Arithmetic Operations

- Use `$()` for integer math operators: `+, -, *, /, % (modulus)`

```
sum=$((5 + 3))
echo "5 + 3 = $sum"
```

3.6 Basic Conditionals

- Syntax: if [condition]; then ... fi
- Comparison:
 - Numbers: -eq, -ne, -lt, -gt
 - Strings: =, !=
 - Files: -e, -f, -d

```
if [ "$1" -gt 10 ]; then
  echo "Greater than 10"
else
  echo "10 or less"
fi
```

3.7 Loops

For loop:

```
for i in 1 2 3; do
  echo "Number $i"
done
```

While loop:

```
count=1
while [ $count -le 5 ]; do
  echo "Count: $count"
  ((count++))
done
```

3.8 Exit Status

- \$? contains exit status of last command (0 = success)
- exit ends the script with a given status

```
grep "pattern" file.txt
if [ $? -eq 0 ]; then
  echo "Pattern found"
else
  echo "Pattern not found"
fi
```

3.9 Basic File Operations

Read file line by line:

```
while read line; do
    echo "Line: $line"
done < input.txt
```

Write to file:

```
echo "New content" > output.txt
```

3.10 Functions

- **Definition:** `function_name() { commands; }`
- **Parameters:** `$1, $2, etc.`
- **Invocation:** `function_name arg1 arg2`

```
greet() {
    echo "Hello, $1"
}
greet "Alice"
```

4 Special Variables in Bash Scripting

Bash provides several special variables that are automatically set by the shell and provide useful information about the script's environment and execution state. Here's a comprehensive guide:

4.1 Positional Parameters

- \$0 - Name of the script being executed
- \$1 to \$9 - First 9 arguments passed to the script
- \${10}, ... - Arguments beyond the 9th (requires curly braces)

```
#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
```

4.2 Script Status Variables

- \$ - Number of arguments passed to the script
- \$* - All arguments as a single string ("12 ...")
- \$@ - All arguments as separate strings ("1" "2" ...)
- \$? - Exit status of the last executed command (0 = success)
- \$\$ - Process ID (PID) of the current shell
- \$! - PID of the last background command

```
echo "Total arguments: $#"
echo "All arguments: $*"
echo "Last command exit status: $?"
echo "Current PID: $$"
```

4.3 Special Shell Variables

- \$ - Last argument of the previous command \$- - Current shell options / flags
- \$IFS - Input Field Separator (default: space, tab, newline)

```
echo "Last argument of previous command: ${_}"
echo "Shell options: ${-}"
```

4.4 Example Usage

```
#!/bin/bash
# special_vars.sh

echo "Running script: $0"
echo "Processing $# arguments"

for arg in "$@"; do
    echo "Argument: $arg"
done

if [ $? -eq 0 ]; then
    echo "Previous command succeeded"
fi

echo "This script's PID: $$"
```

4.5 Key Differences Between \$* and \$@

Variable	Without Quotes	With Quotes
\$*	Single string	Single string
\$@	Separate strings	Preserves arguments

```
# Demonstrate difference
for item in "$*"; do echo "$item"; done  # Single iteration
for item in "$@"; do echo "$item"; done  # One per argument
```

4.6 Best Practices

- Always quote "\$@" when iterating over arguments
- Use \${10} for arguments beyond 9th position
- Check \$? after critical commands
- Use \$\$ for creating temporary files with unique names

4.7 Common Exit Status Codes

- 0: Success
- 1: General error
- 2: Misuse of shell builtins
- 127: Command not found
- 130: Script terminated by Ctrl+C (SIGINT)

5 Input/Output in Bash Scripting

Bash provides several powerful ways to handle input and output operations. Here's a comprehensive guide to I/O in Bash scripts:

5.1 Standard Output (stdout)

echo command - Basic output

```
echo "Hello World"          # Basic output
echo -e "Line1\nLine2"      # Enable interpretation of backslash escapes
echo -n "No newline"        # Suppress trailing newline
```

printf command - Formatted output

```
printf "Name: %s\nAge: %d\n" "Alice" 25
printf "Decimal: %d\nHex: %x\nFloat: %.2f\n" 100 100 3.14159
```

5.2 Standard Input (stdin)

Reading input

```
read name                # Basic read
read -p "Enter your name: " name # With prompt
read -s -p "Password: " pass # Silent input (for passwords)
read -t 5 -p "Timeout: " input # Timeout after 5 seconds
```

Reading multiple values

```
read first last           # Reads two words
read -a array              # Reads into an array
```

5.3 File Operations

Reading files

```
while read line; do          # Read line by line
    echo "$line"
done < file.txt

mapfile -t lines < file.txt # Read entire file into array
```

Writing to files

```
echo "content" > file.txt   # Overwrite file
echo "more" >> file.txt    # Append to file
```

5.4 Redirection

Basic redirection

```
command > file.txt         # Redirect stdout to file (overwrite)
command >> file.txt        # Redirect stdout to file (append)
command < file.txt          # Redirect file to stdin
```

Error redirection

```
command 2> error.log      # Redirect stderr
command &> output.log     # Redirect both stdout and stderr
command 2>&1              # Redirect stderr to stdout
```

Advanced redirection

```
command > >(tee log.txt)    # Process substitution
command | tee log.txt       # Output to both console and file
```

5.5 Here Documents & Strings

Here document (ii)

```
cat << EOF
This is a multi-line
input that continues
until the EOF delimiter
EOF
```

Here string (iii)

```
grep "pattern" <<< "$variable"
```

5.6 Special File Descriptors

Descriptor	Purpose
0	stdin
1	stdout
2	stderr
3-9	Custom streams

Example:

```
exec 3> custom.log  # Open fd 3 for writing
echo "Message" >&3  # Write to fd 3
exec 3>&-            # Close fd 3
```

5.7 Practical Examples

Logging script output

```
#!/bin/bash
exec > >(tee -a script.log) 2>&1
echo "This will be logged to script.log and console"
```

Prompt with validation

```
while true; do
    read -p "Enter age (0-120): " age
    [[ $age =~ ^[0-9]+\$ ]] && ((age >= 0 && age <= 120)) && break
    echo "Invalid input!"
done
```

CSV file processing

```
while IFS=',' read -r col1 col2 col3; do
    echo "Column 1: $col1"
    echo "Column 2: $col2"
done < data.csv
```

5.8 Best Practices

- Always quote variables when echoing ("\$var")
- Use `printf` for complex formatting
- Prefer <<< here strings over temporary files
- Close custom file descriptors when done
- Use `-r` with `read` to preserve backslashes
- Consider `set -o errexit` to exit on command failure

6 Conditionals in Bash Scripting

Conditionals allow your Bash scripts to make decisions and execute different code blocks based on various conditions. Here's a comprehensive guide to using conditionals in Bash:

6.1 Basic if-then-else Structure

```
if [ condition ]; then
    # commands if true
elif [ another_condition ]; then
    # commands if elif true
else
    # commands if all false
fi
```

6.2 Test Conditions (Square Brackets [])

The [] is actually a command (synonym for `test`) that evaluates conditions.

File Tests

```
if [ -e "file.txt" ]; then      # File exists
if [ -f "file.txt" ]; then      # Regular file
if [ -d "dir" ]; then          # Directory exists
if [ -s "file.txt" ]; then      # File exists and is not empty
if [ -r "file.txt" ]; then      # File is readable
if [ -w "file.txt" ]; then      # File is writable
if [ -x "file.txt" ]; then      # File is executable
if [ "file1" -nt "file2" ]; then # File1 is newer than file2
if [ "file1" -ot "file2" ]; then # File1 is older than file2
```

String Comparisons

```
if [ "$str1" = "$str2" ]; then  # Strings are equal
if [ "$str1" != "$str2" ]; then  # Strings are not equal
if [ -z "$str" ]; then         # String is empty
if [ -n "$str" ]; then         # String is not empty
if [[ "$str1" < "$str2" ]]; then # str1 sorts before str2 (lexicographically)
```

Numeric Comparisons

```
if [ "$a" -eq "$b" ]; then      # Equal (==)
if [ "$a" -ne "$b" ]; then      # Not equal (!=)
if [ "$a" -lt "$b" ]; then      # Less than (<)
if [ "$a" -le "$b" ]; then      # Less than or equal (<=)
if [ "$a" -gt "$b" ]; then      # Greater than (>)
if [ "$a" -ge "$b" ]; then      # Greater than or equal (>=)
```

6.3 Advanced Conditional Expressions ([[]])

The double square brackets [[]] offer more features than [].

```
if [[ $var == "value" ]]; then      # Pattern matching
if [[ $var =~ ^regex$ ]]; then      # Regular expression matching
if [[ $var1 == "$var2" ]]; then      # Safer string comparison
if [[ -e "$file" && -r "$file" ]]; then # Multiple conditions
```

6.4 Logical Operators

```
if [ "$a" -eq 1 ] && [ "$b" -eq 2 ]; then # AND
if [ "$a" -eq 1 ] || [ "$b" -eq 2 ]; then # OR
if ! [ "$a" -eq 1 ]; then                 # NOT

# Alternative syntax with [[ ]]
if [[ $a -eq 1 && $b -eq 2 ]]; then
if [[ $a -eq 1 || $b -eq 2 ]]; then
if [[ ! $a -eq 1 ]]; then
```

6.5 Case Statements

For matching against multiple patterns:

```
case "$variable" in
  pattern1)
    # commands for pattern1
    ;;
  pattern2|pattern3)
    # commands for pattern2 or pattern3
    ;;
  *)
    # default case
    ;;
esac
```

Example:

```
read -p "Enter y/n: " answer
case "$answer" in
  [Yy] | "yes") echo "You agreed" ;;
  [Nn] | "no") echo "You declined" ;;
  *) echo "Invalid input" ;;
esac
```

6.6 Arithmetic Conditions

```
if (( $a + 5 > $b )); then      # Arithmetic evaluation
  echo "a + 5 is greater than b"
fi
```

6.7 Combining Conditions

```
if [ "$user" = "admin" ] && { [ "$day" = "Monday" ] || [ "$day" = "Wednesday" ] }; then
  echo "Admin access granted on Monday or Wednesday"
fi
```

6.8 Common Pitfalls

1. **Spacing:** Always put spaces around brackets and operators
 - Correct: ["\$a" -eq "\$b"]
 - Wrong: ["\$a"-eq"\$b"]
2. **Quoting variables:** Always quote variables to prevent word splitting
 - Correct: ["\$var" = "value"]
 - Wrong: [\$var = "value"]
3. **String vs numeric comparisons:**
 - Strings: =, !=
 - Numbers: -eq, -ne, etc.

6.9 Practical Examples

Check if user is root

```
if [ "$(id -u)" -eq 0 ]; then
    echo "Running as root"
else
    echo "Run this script as root" >&2
    exit 1
fi
```

Check file extension

```
filename="document.txt"
if [[ "$filename" =~ \.txt$ ]]; then
    echo "Text file detected"
fi
```

Validate input

```
read -p "Enter a number (1-100): " num
if ! [[ "$num" =~ ^[0-9]+$ ]] || [ "$num" -lt 1 ] || [ "$num" -gt 100 ]; then
    echo "Invalid number!" >&2
    exit 1
fi
```

Conditionals are fundamental to making your Bash scripts intelligent and responsive to different situations. The key is to choose the right type of conditional structure for your specific use case and to be mindful of the syntax requirements.

7 Loops in Bash Scripting

Loops are essential for automating repetitive tasks in Bash. Here are the main loop constructs with practical examples:

7.1 For Loops

Basic for loop (iterating through list)

```
for fruit in apple banana orange; do
    echo "I like $fruit"
done
```

C-style for loop

```
for ((i=1; i<=5; i++)); do
    echo "Count: $i"
done
```

Iterating through files

```
for file in *.txt; do
    echo "Processing $file"
    wc -l "$file"
done
```

Iterating through command output

```
for user in $(cut -d: -f1 /etc/passwd | head -5); do
    echo "User: $user"
done
```

7.2 While Loops

Basic while loop

```
count=1
while [ $count -le 5 ]; do
    echo "Count: $count"
    ((count++))
done
```

Reading file line by line

```
while IFS= read -r line; do
    echo "Line: $line"
done < /etc/hosts
```

Infinite loop with break

```
while true; do
    read -p "Enter command (q to quit): " cmd
    [ "$cmd" = "q" ] && break
    echo "You entered: $cmd"
done
```

7.3 Until Loops

Runs until condition becomes true

```
attempt=1
until ping -c1 example.com &>/dev/null; do
    echo "Attempt $attempt: Waiting for connection..."
    ((attempt++))
    sleep 1
done
echo "Connection established!"
```

7.4 Loop Control Statements

break - exit loop

```
for i in {1..10}; do
    if [ $i -eq 5 ]; then
        break
    fi
    echo "$i"
done
```

continue - skip current iteration

```
for i in {1..5}; do
    [ $i -eq 3 ] && continue
    echo "$i"
done
```

7.5 Practical Examples

Backup files modified today

```
backup_dir="/backups/$(date +%Y-%m-%d)"
mkdir -p "$backup_dir"

for file in /data/*; do
    if [ "$(date -r "$file" +%Y-%m-%d)" = "$(date +%Y-%m-%d)" ]; then
        cp -v "$file" "$backup_dir"
    fi
done
```

Monitor disk space

```
while true; do
    clear
    df -h
    sleep 5
done
```

Password generator

```
until [[ "$password" =~ [A-Z] && "$password" =~ [0-9] ]]; do
    password=$(openssl rand -base64 12)
done
echo "Secure password: $password"
```

7.6 Best Practices

- Always quote variables in loops ("\$var")
- Use IFS= and -r when reading files line by line
- For arithmetic operations, prefer (()) over let or expr
- Consider adding sleep in infinite loops to reduce CPU usage
- Use meaningful variable names in nested loops
- Prefer [[]] over [] for pattern matching

7.7 Nested Loops Example

```
for i in {1..3}; do
    echo "Outer loop: $i"
    for j in {a..c}; do
        echo "  Inner loop: $j"
    done
done
```

7.8 Special Loop Forms

Process substitution loop

```
while read -r line; do
    echo "Processing: $line"
done < <(grep "error" /var/log/syslog)
```

Loop through array

```
colors=("red" "green" "blue")
for color in "${colors[@]}"; do
    echo "Color: $color"
done
```

Loops are powerful tools that can handle everything from simple iterations to complex automation tasks in Bash scripting.

8 Arrays in Bash Scripting

Arrays allow you to store and manipulate multiple values in Bash. Here's a comprehensive guide to using arrays effectively:

8.1 Array Basics

Creating Arrays

```
# Indexed array (default)
fruits=("apple" "banana" "cherry")

# Explicit indexed array declaration
declare -a colors=("red" "green" "blue")

# Associative array (Bash 4+)
declare -A user=([name]="Alice" [age]=25 [city]="Paris")
```

Accessing Elements

```
echo ${fruits[0]}      # apple (index starts at 0)
echo ${user["name"]} # Alice
```

8.2 Working with Indexed Arrays

Initialization Methods

```
numbers=(1 2 3 4 5)
names=([0]="Alice" [3]="Bob") # Sparse array
letters=({a..z})             # Range expansion
```

Common Operations

```
# Add elements
fruits+=("orange" "grape")

# Modify element
fruits[1]="kiwi"

# Get array length
echo ${#fruits[@]}

# Get all elements
echo ${fruits[@]}

# Get all indices
echo ${!fruits[@]}

# Slice array
echo ${fruits[@]:1:2} # elements 1-2
```

8.3 Working with Associative Arrays (Bash 4+)

Declaration and Usage

```
declare -A employee=(
    ["id"]=1001
    ["name"]="John Doe"
    ["department"]="Engineering"
```

```
)
# Add/modify elements
employee["salary"]=85000
employee["department"]="IT"

# Access all keys
echo ${!employee[@]}

# Access all values
echo ${employee[@]}
```

8.4 Iterating Through Arrays

Indexed Arrays

```
# By index
for i in "${!fruits[@]}"; do
    echo "$i: ${fruits[$i]}"
done

# By value
for fruit in "${fruits[@]}"; do
    echo "$fruit"
done
```

Associative Arrays

```
for key in "${!employee[@]}"; do
    echo "$key: ${employee[$key]}"
done
```

8.5 Array Manipulation

Copying Arrays

```
copy=("${fruits[@]}")
```

Concatenating Arrays

```
combined=("${fruits[@]}" "${colors[@]}")
```

Removing Elements

```
unset fruits[1]      # Remove banana
unset user["city"]   # Remove city from associative array
```

Sorting Arrays

```
sorted=$(printf "%s\n" "${fruits[@]}" | sort)
```

8.6 Practical Examples

1. Processing command output

```
# Store process IDs in array
pids=$(pgrep -u $USER)

echo "Your running processes:"
for pid in "${pids[@]}"; do
    echo "- $pid: $(ps -p $pid -o comm)"
done
```

2. Configuration management

```
declare -A config
while IFS='=' read -r key value; do
    config["$key"]="$value"
done < config.properties

echo "Database host: ${config[db_host]}"
```

3. Matrix operations

```
declare -A matrix
matrix[0,0]=1; matrix[0,1]=2
matrix[1,0]=3; matrix[1,1]=4

for i in 0 1; do
    for j in 0 1; do
        printf "%s " ${matrix[$i,$j]}
    done
    echo
done
```

8.7 Best Practices

- Always quote array expansions: "\${array[@]}"
- Use `declare -A` for associative arrays (Bash 4+ required)
- Prefer indexed arrays for ordered lists
- Use associative arrays for key-value pairs
- Avoid sparse arrays when possible
- Consider using `mapfile` for reading files into arrays

8.8 Common Pitfalls

- Forgetting that indices start at 0
- Mixing array types (indexed vs associative)
- Assuming arrays are passed to functions (they aren't - pass by reference)
- Modifying arrays while iterating through them

Arrays are powerful tools in Bash scripting that can significantly simplify working with collections of data. They're particularly useful for configuration management, data processing, and when working with structured information.

9 Functions in Bash Scripting

Functions in Bash allow you to group commands into reusable blocks of code. Here's a comprehensive guide to using functions effectively:

9.1 Basic Function Syntax

Definition Methods

```
# Method 1
function_name() {
    commands
}

# Method 2 (POSIX compliant)
function function_name {
    commands
}
```

Simple Example

```
greet() {
    echo "Hello, $1!"
}

greet "Alice" # Output: Hello, Alice!
```

9.2 Function Parameters

Accessing Arguments

```
show_details() {
    echo "Name: $1"
    echo "Age: $2"
    echo "All arguments: $@"
    echo "Argument count: $#"
}

show_details "Bob" 25 "New York"
```

Default Values

```
connect() {
    local host=${1:-"localhost"}
    local port=${2:-"8080"}
    echo "Connecting to $host:$port"
}

connect # Uses defaults
connect "example.com" 443
```

9.3 Return Values

Exit Status (0–255)

```
is_even() {
    if (( $1 % 2 == 0 )); then
        return 0 # Success
    else
```

```

        return 1 # Failure
    fi
}

is_even 4 && echo "Even" || echo "Odd"

```

Returning Data

```

get_date() {
    echo $(date +%F)
}

today=$(get_date)

```

9.4 Variable Scope

Local Variables

```

calculate() {
    local result=$(( $1 + $2 ))
    echo $result
}

sum=$(calculate 5 3)

```

Global Variables

```

counter=0

increment() {
    ((counter++))
}

increment
echo $counter # Output: 1

```

9.5 Advanced Function Features

Passing Arrays

```

process_array() {
    local -n arr=$1 # Nameref (Bash 4.3+)
    for item in "${arr[@]}"; do
        echo "Processing: $item"
    done
}

fruits=("apple" "banana")
process_array fruits

```

Recursive Functions

```

factorial() {
    if (( $1 <= 1 )); then
        echo 1
    else
        local prev=$(factorial $(( $1 - 1 )))
        echo $(( $1 * prev ))
    fi
}

```

```

}

factorial 5 # Output: 120

```

9.6 Practical Examples

1. File Processor

```

process_files() {
    local dir=$1
    local ext=$2
    for file in "$dir"/*."$ext"; do
        [ -f "$file" ] || continue
        echo "Processing $file"
        # Add processing commands here
    done
}

process_files "/data" "csv"

```

2. Input Validator

```

validate_input() {
    while true; do
        read -p "$1" input
        if [[ "$input" =~ $2 ]]; then
            echo "$input"
            return 0
        fi
        echo "Invalid input, please try again." >&2
    done
}

email=$(validate_input "Enter email: " "[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$")

```

3. Logger Function

```

log() {
    local level=$1
    local message=$2
    local timestamp=$(date "+%Y-%m-%d %H:%M:%S")
    echo "[${timestamp}] [${level}] ${message}" >> "$LOG_FILE"

    if [[ "${level}" == "ERROR" ]]; then
        echo "$message" >&2
    fi
}

log "INFO" "Process started"
log "ERROR" "File not found"

```

9.7 Best Practices

- Use `local` for function-scoped variables
- Always validate function parameters
- Use meaningful function names
- Keep functions small and focused

- Document functions with comments
- Return proper exit codes (0 for success)
- Use `readonly -f` to protect important functions

9.8 Common Pitfalls

- Forgetting that parameters are positional (`$1`, `$2`, etc.)
- Modifying global variables unintentionally
- Not handling errors within functions
- Assuming function variables persist after execution
- Overcomplicating functions (they should do one thing well)

Functions are essential for writing modular, maintainable Bash scripts. They help reduce code duplication and make your scripts more organized and readable.

10 String Manipulation in Bash Scripting

Bash provides powerful string manipulation capabilities that allow you to transform, extract, and analyze text efficiently. Here's a comprehensive guide to working with strings in Bash.

10.1 Basic String Operations

Variable Assignment

```
str="Hello World"
```

String Length

```
echo ${#str} # Output: 11
```

Concatenation

```
str1="Hello"
str2="World"
combined="$str1 $str2" # "Hello World"
combined=$str1$str2 # "HelloWorld"
```

10.2 Substring Operations

Extract Substring

```
 ${string:position:length}

str="abcdefg"
echo ${str:2:3} # Output: cde (starts at position 2, length 3)
```

From Beginning/End

```
echo ${str: -4} # Last 4 characters: efg
echo ${str:3} # From position 3 to end: defgh
```

10.3 Pattern Matching and Replacement

Remove Matching Prefix

```
 ${variable#pattern} # Shortest match
 ${variable##pattern} # Longest match

path="/usr/local/bin"
echo ${path#/usr} # "/local/bin"
echo ${path##*/} # "bin" (basename)
```

Remove Matching Suffix

```
 ${variable%pattern} # Shortest match
 ${variable%%pattern} # Longest match

file="archive.tar.gz"
echo ${file%.*} # "archive.tar"
echo ${file%%.*} # "archive"
```

Replace Patterns

```

${variable/pattern/replacement}  # First match
${variable//pattern/replacement} # All matches

str="hello hello hello"
echo ${str/hello/Hi}      # "Hi hello hello"
echo ${str//hello/Hi}     # "Hi Hi Hi"

```

Case Conversion (Bash 4+)

```

str="Hello World"
echo ${str^^} # Uppercase: "HELLO WORLD"
echo ${str,,} # Lowercase: "hello world"
echo ${str^}  # Capitalize first letter: "Hello World"

```

10.4 String Splitting and Joining

Split by Delimiter

```

IFS=: read -ra parts <<< "one:two:three"
echo "${parts[1]}" # "two"

```

Join Array Elements

```

arr=("one" "two" "three")
joined=$(IFS=,; echo "${arr[*]}")
echo "$joined" # "one,two,three"

```

10.5 String Validation and Testing

Check if Empty

```

if [ -z "$str" ]; then
    echo "String is empty"
fi

```

Check if Contains Substring

```

if [[ $str == *"World"* ]]; then
    echo "Contains 'World'"
fi

```

Regex Matching (Bash 3.2+)

```

if [[ $str =~ ^Hello ]]; then
    echo "Starts with Hello"
fi

```

10.6 Advanced String Processing

Remove Whitespace

```

str=" hello "
echo "${str}" | xargs      # Trim both sides: "hello"
echo "${str%${str##*[![:space:]]}}" # Trim right
echo "${str#${str%*[![:space:]]}}" # Trim left

```

URL Encoding/Decoding

```
urlencode() {
    local string="${1}"
    local strlen=${#string}
    local encoded=""

    for (( pos=0 ; pos<strlen ; pos++ )); do
        c=${string:$pos:1}
        case "$c" in
            [-_.a-zA-Z0-9]) encoded+="$c" ;;
            *) printf -v encoded '%/%02X' "'$c" ;;
        esac
    done
    echo "$encoded"
}

urldecode() {
    local url_encoded="${1//+/ }"
    printf '%b' "${url_encoded//%/\\x}"
}
```

10.7 Practical Examples

Filename Manipulation

```
file="/path/to/file.txt"
filename="${file##*/}"           # "file.txt"
extension="${file##*.}"          # "txt"
basename="${filename%.*}"        # "file"
dirname="${file%/*}"             # "/path/to"
```

CSV Processing

```
process_csv() {
    local line="John,Doe,30,New York"
    IFS=',' read -ra fields <<< "$line"
    echo "Name: ${fields[0]} ${fields[1]}"
    echo "Age: ${fields[2]}"
    echo "City: ${fields[3]}"
}
```

Password Generator

```
generate_password() {
    local length=${1:-12}
    local chars='A-Za-z0-9!@#$%^&*()'
    LC_ALL=C tr -dc "$chars" < /dev/urandom | head -c "$length"
    echo
}
```

10.8 Best Practices

- Always quote variables to preserve spaces and special characters
- Use `${var:-default}` for default values
- Prefer built-in string operations over external commands
- For complex text processing, consider using `awk` or `sed`

- Use `declare -l` or `declare -u` for case conversion (Bash 4+)

String manipulation is fundamental in shell scripting, and mastering these techniques will enable you to handle text processing tasks efficiently in Bash.

11 Arithmetic in Bash Scripting

Bash provides several ways to perform arithmetic operations, each with its own use cases and limitations. Here's a comprehensive guide to arithmetic in Bash:

11.1 Basic Arithmetic Methods

11.1.1 Double Parentheses ((()))

```
(( result = 5 + 3 ))
echo $result # Output: 8

# Can be used directly in conditionals
if (( 5 > 3 )); then
    echo "Five is greater than three"
fi
```

11.1.2 Dollar Double Parentheses \${(())}

```
result=$((5 + 3))
echo $result # Output: 8

# With variables
a=5
b=3
echo $((a + b)) # Output: 8
```

11.1.3 let Command

```
let "result = 5 + 3"
echo $result # Output: 8

# Multiple operations
let "a=5, b=3, result=a+b"
```

11.1.4 expr Command (legacy)

```
result=$(expr 5 + 3)
echo $result # Output: 8
```

11.2 Arithmetic Operators

11.2.1 Basic Operators

```
echo $((10 + 2)) # Addition: 12
echo $((10 - 2)) # Subtraction: 8
echo $((10 * 2)) # Multiplication: 20
echo $((10 / 2)) # Integer division: 5
echo $((10 % 3)) # Modulus: 1
echo $((10 ** 2)) # Exponentiation: 100
```

11.2.2 Bitwise Operators

```
echo $((5 & 3)) # AND: 1
echo $((5 | 3)) # OR: 7
echo $((5 ^ 3)) # XOR: 6
echo $((~5)) # NOT: -6
```

```
echo $((5 << 1))  # Left shift: 10
echo $((5 >> 1))  # Right shift: 2
```

11.2.3 Assignment Operators

```
((a = 5))
((a += 2))  # a = 7
((a -= 1))  # a = 6
((a *= 2))  # a = 12
((a /= 3))  # a = 4
((a %= 3))  # a = 1
```

11.3 Number Bases

11.3.1 Different Base Representation

```
echo $((0xff))      # Hexadecimal: 255
echo $((16#ff))    # Hexadecimal: 255
echo $((2#1010))   # Binary: 10
echo $((8#12))     # Octal: 10
```

11.3.2 Base Conversion

```
printf "%d\n" 0xff      # Hex to decimal: 255
printf "%x\n" 255       # Decimal to hex: ff
printf "%o\n" 8         # Decimal to octal: 10
```

11.4 Floating-Point Arithmetic

11.4.1 Using bc (for floating-point)

```
result=$(echo "scale=2; 10/3" | bc)
echo $result  # Output: 3.33

# More complex example
area=$(echo "scale=2; 3.14 * 5 * 5" | bc)
echo $area    # Output: 78.50
```

11.4.2 Using awk

```
result=$(awk 'BEGIN {print 10/3}')
echo $result  # Output: 3.33333
```

11.5 Increment/Decrement Operations

11.5.1 Postfix and Prefix

```
a=5
echo $((a++))  # Output: 5 (then a becomes 6)
echo $((++a))  # Output: 7 (a is now 7)
echo $((a--))  # Output: 7 (then a becomes 6)
echo $((--a))  # Output: 5 (a is now 5)
```

11.6 Practical Examples

11.6.1 Simple Calculator

```
read -p "Enter first number: " num1
read -p "Enter second number: " num2

echo "Addition: $((num1 + num2))"
echo "Subtraction: $((num1 - num2))"
echo "Multiplication: $((num1 * num2))"
echo "Division: $((num1 / num2))"
echo "Modulus: $((num1 % num2))"
```

11.6.2 Temperature Conversion

```
celsius_to_fahrenheit() {
    echo "scale=2; \$1 * 9/5 + 32" | bc
}

fahrenheit_to_celsius() {
    echo "scale=2; (\$1 - 32) * 5/9" | bc
}

echo "20°C is $(celsius_to_fahrenheit 20)°F"
echo "68°F is $(fahrenheit_to_celsius 68)°C"
```

11.6.3 Prime Number Check

```
is_prime() {
    local num=$1
    for ((i=2; i*i<=num; i++)); do
        if ((num % i == 0)); then
            return 1 # Not prime
        fi
    done
    return 0 # Prime
}

is_prime 17 && echo "Prime" || echo "Not prime"
```

11.7 Best Practices

- Use `(())` for integer arithmetic (most efficient)
- Use `bc` for floating-point calculations
- Always initialize variables before arithmetic operations
- For complex math, consider using external tools like `awk` or `bc`
- Be aware of integer division behavior (truncates decimal part)
- Use `$[]` only for backward compatibility (deprecated)

11.8 Common Pitfalls

- Forgetting that Bash only handles integer arithmetic natively
- Not quoting variables when using `expr`
- Using spaces incorrectly in arithmetic expressions

- Forgetting that division truncates ($10/3 = 3$, not 3.333)
- Not handling division by zero cases

Arithmetic operations in Bash are powerful for integer calculations, and when combined with tools like **bc** or **awk**, can handle most mathematical needs in shell scripting.

12 File Operations in Bash Scripting

Bash provides powerful tools for working with files and directories. Here's a comprehensive guide to file operations in Bash:

12.1 File Testing and Inspection

12.1.1 Test Operators

```
[[ -e "file.txt" ]]      # File exists
[[ -f "file.txt" ]]      # Regular file
[[ -d "dir" ]]           # Directory exists
[[ -s "file.txt" ]]      # File exists and is not empty
[[ -r "file.txt" ]]      # File is readable
[[ -w "file.txt" ]]      # File is writable
[[ -x "file.txt" ]]      # File is executable
[[ "f1" -nt "f2" ]]      # f1 is newer than f2
[[ "f1" -ot "f2" ]]      # f1 is older than f2
```

12.1.2 Get File Information

```
stat -c "%n %s bytes" file.txt    # Name and size
file --mime-type document.pdf     # MIME type
wc -l file.txt                   # Line count
md5sum file.txt                  # Calculate MD5 hash
```

12.2 File Reading and Writing

12.2.1 Read Entire File

```
content=$(<file.txt)          # Fastest method
content=$(cat file.txt)        # Alternative
```

12.2.2 Read Line by Line

```
while IFS= read -r line; do
    echo "Line: $line"
done < file.txt
```

12.2.3 Write to File

```
echo "content" > file.txt       # Overwrite
echo "more" >> file.txt        # Append
printf "%s\n" "line1" "line2" > file.txt
```

12.2.4 Here Documents

```
cat > config.conf << EOF
[settings]
host=example.com
port=8080
EOF
```

12.3 File Manipulation

12.3.1 Copy, Move, Rename

```
cp source.txt dest.txt
cp -r dir1/ dir2/           # Recursive copy
mv old.txt new.txt          # Rename or move
```

12.3.2 Create and Remove

```
touch newfile.txt            # Create empty file
mkdir -p path/to/dir        # Create directory (with parents)
rm file.txt                 # Remove file
rm -r dir/                  # Remove directory recursively
```

12.3.3 Find Files

```
find . -name "*.txt"        # Find by name
find . -type f -mtime -7    # Modified in last 7 days
find . -size +1M            # Files larger than 1MB
```

12.4 File Permissions

12.4.1 Change Permissions

```
chmod 755 script.sh          # rwxr-xr-x
chmod +x script.sh          # Add execute
chmod u=rw,g=r,o= file.txt  # User:rw, Group:r, Others:none
```

12.4.2 Change Ownership

```
chown user:group file.txt
chown -R user:group dir/    # Recursive
```

12.5 Advanced Operations

12.5.1 Temporary Files

```
tempfile=$(mktemp /tmp/example.XXXXXX)
trap 'rm -f "$tempfile"' EXIT  # Cleanup on exit
```

12.5.2 File Locking

```
((
  flock -x 200
  # Critical section
  echo "Writing" >> file.log
) 200>lockfile
```

12.5.3 Compare Files

```
diff file1.txt file2.txt
cmp -b file1.bin file2.bin   # Binary comparison
```

12.6 Practical Examples

12.6.1 Backup Script

```
backup_files() {
    local src_dir=$1
    local dest_dir=$2
    local timestamp=$(date +%Y%m%d_%H%M%S)
    local backup_file="${dest_dir}/backup_${timestamp}.tar.gz"

    tar -czf "$backup_file" -C "$src_dir" .
    echo "Backup created: $backup_file"
}

backup_files "/data" "/backups"
```

12.6.2 Log Rotator

```
rotate_logs() {
    local logfile=$1
    local max_backups=5

    for ((i=max_backups; i>0; i--)); do
        [[ -f "${logfile}.$i" ]] && mv "${logfile}.$i" "${logfile}.${((i+1))}"
    done
    [[ -f "$logfile" ]] && mv "$logfile" "${logfile}.1"
    touch "$logfile"
}

rotate_logs "/var/log/app.log"
```

12.6.3 File Search and Process

```
search_and_process() {
    local pattern=$1
    local action=$2

    while IFS= read -r -d '\0' file; do
        case $action in
            "delete") rm -v "$file" ;;
            "compress") gzip -v "$file" ;;
            *) echo "Processing $file" ;;
        esac
    done < <(find . -name "$pattern" -print0)
}

search_and_process "*.tmp" "delete"
```

12.7 Best Practices

- Always quote file paths to handle spaces
- Use `-r` with `read` to preserve backslashes
- Prefer `find -print0` with `read -d ''` for handling strange filenames
- Check file existence before operations
- Use `set -o noclobber` to prevent accidental overwrites
- Implement proper error handling
- Clean up temporary files

12.8 Common Pitfalls

- Not handling filenames with spaces or special characters
- Forgetting to check if files exist before operations
- Not properly closing file descriptors
- Race conditions in file operations
- Assuming a file's content based on its extension
- Not setting proper permissions on created files

These file operations form the foundation for most Bash scripting tasks involving files and directories. Would you like me to elaborate on any specific aspect?

13 Process Control in Bash Scripting

Process control is essential for managing and interacting with system processes in Bash scripts. Here's a comprehensive guide to process control techniques:

13.1 Process Basics

13.1.1 Running Processes

```
# Run in foreground (default)
command

# Run in background
command &

# Run with nohup (persists after logout)
nohup command &
```

13.1.2 Process Status

```
ps                      # Show current processes
ps aux                 # Show all processes
pgrep -l "process_name" # Find process by name
pidof program_name      # Get PID of running program
```

13.2 Job Control

13.2.1 Managing Background Jobs

```
command &                # Start in background
jobs                   # List background jobs
fg %1                  # Bring job 1 to foreground
bg %2                  # Resume job 2 in background
kill %3                # Terminate job 3
```

13.2.2 Job Notification

```
set -b                  # Immediate job notification
notify                 # Notify when current job completes
```

13.3 Process Termination

13.3.1 Killing Processes

```
kill PID                # Terminate process (SIGTERM)
kill -9 PID              # Force kill (SIGKILL)
killall process_name     # Kill all processes by name
pkill "pattern"          # Kill by pattern
```

13.3.2 Graceful Shutdown

```
# Send TERM signal, wait, then force kill
kill PID || sleep 5 && kill -9 PID
```

13.4 Process Monitoring

13.4.1 Watch Processes

```
top          # Interactive process viewer
htop         # Enhanced top (if installed)
watch -n 1 'ps aux | grep httpd' # Refresh every second
```

13.4.2 Timeout Control

```
timeout 10s slow_command # Kill after 10 seconds
timeout -k 5 10s command # Send SIGTERM after 10s, SIGKILL after 5s
```

13.5 Advanced Process Management

13.5.1 Process Substitution

```
diff <(sort file1) <(sort file2)
```

13.5.2 Named Pipes

```
mkfifo mypipe
command1 > mypipe &
command2 < mypipe
```

13.5.3 Process Priority

```
nice -n 10 command      # Run with lower priority
renice 15 -p PID        # Change priority of running process
```

13.6 Practical Examples

13.6.1 Process Monitor Script

```
monitor_process() {
    local process_name=$1
    local max_attempts=3
    local attempt=1

    while (( attempt <= max_attempts )); do
        if pgrep -x "$process_name" >/dev/null; then
            echo "$process_name is running"
            return 0
        else
            echo "Attempt $attempt: $process_name not running"
            ((attempt++))
            sleep 2
        fi
    done

    echo "Error: $process_name failed to start"
    return 1
}

monitor_process "nginx"
```

13.6.2 Parallel Processing

```
# Run up to 4 parallel processes
for item in {1..10}; do
    ((i=i%4)); ((i++==0)) && wait
    process_item "$item" &
done
wait # Wait for all background processes
```

13.6.3 Service Manager

```
manage_service() {
    case "$1" in
        start)
            nohup /usr/local/bin/service_daemon &> /var/log/service.log &
            echo $! > /var/run/service.pid
            ;;
        stop)
            kill -TERM $(cat /var/run/service.pid)
            rm /var/run/service.pid
            ;;
        restart)
            manage_service stop
            sleep 2
            manage_service start
            ;;
        *)
            echo "Usage: $0 {start|stop|restart}"
            exit 1
    esac
}

manage_service "$1"
```

13.7 Best Practices

- Always clean up background processes
- Use proper signal handling (`trap`)
- Implement timeout for critical operations
- Check process status before killing
- Use process substitution instead of temporary files when possible
- Consider using `screen` or `tmux` for long-running processes

13.8 Common Pitfalls

- Zombie processes from improper cleanup
- Race conditions in process management
- Not handling signals properly
- Assuming a process died without verification
- Resource leaks from too many parallel processes
- Forgetting to wait for background processes

Process control is fundamental for writing robust Bash scripts that interact with system processes. Would you like me to elaborate on any specific aspect?

14 Advanced Features in Bash Scripting

Bash offers powerful advanced features that enable sophisticated scripting capabilities. Here's a comprehensive guide to these advanced techniques:

14.1 Arrays and Associative Arrays

14.1.1 Indexed Arrays

```
files=("file1.txt" "file2.txt" "file3.txt")
echo ${files[1]}          # file2.txt (0-based index)
echo ${files[@]}           # All elements
echo ${#files[@]}          # Number of elements

# Modify arrays
files+=("file4.txt")      # Append element
unset files[1]              # Remove element
```

14.1.2 Associative Arrays (Bash 4+)

```
declare -A user
user=(["name"]="Alice" ["age"]=25 ["city"]="Paris")
echo ${user["name"]}        # Alice
echo ${!user[@]}            # All keys
```

14.2 Process Substitution

14.2.1 Compare outputs without temp files

```
diff <(sort file1) <(sort file2)

# Feed multiple inputs
paste <(cut -f1 data.txt) <(cut -f3 data.txt)
```

14.3 Coprocesses

14.3.1 Two-way communication with processes

```
coproc bc
echo "5+3" >&${COPROC[1]}
read result <&${COPROC[0]}
echo $result      # 8
```

14.4 Named Pipes (FIFOs)

14.4.1 Create persistent pipes

```
mkfifo mypipe
processor < mypipe &      # Background process
generator > mypipe         # Feed data
```

14.5 Indirect References

14.5.1 Dynamic variable names

```
var="value"
ref="var"
```

```
echo ${!ref}           # value
```

14.6 Parameter Transformation

14.6.1 Advanced string manipulation

```
 ${var@U}          # Uppercase
 ${var@L}          # Lowercase
 ${var@A}          # Show as assignment
 ${var@P}          # Prompt string expansion
```

14.7 Regular Expressions

14.7.1 Native regex support (Bash 3.2+)

```
if [[ "hello" =~ ^h.+o$ ]]; then
    echo "Pattern matched"
fi

# Capture groups
[[ "date: 2023-01-15" =~ ([0-9]{4}-[0-9]{2}-[0-9]{2}) ]]
echo ${BASH_REMATCH[1]} # 2023-01-15
```

14.8 Advanced I/O Redirection

14.8.1 Custom file descriptors

```
exec 3<> file.txt      # Open for read/write
echo "data" >&3          # Write to fd 3
read line <&3            # Read from fd 3
exec 3>&-              # Close fd 3
```

14.9 Error Handling

14.9.1 Advanced trap usage

```
trap 'cleanup; exit 1' ERR SIGINT SIGTERM

cleanup() {
    echo "Removing temp files..."
    rm -f temp_*
}
```

14.10 Namespace Control

14.10.1 Restricted shells

```
bash --restricted
```

14.10.2 Function namespaces

```
namespace::func() {
    echo "This function is in a namespace"
}
```

14.11 Practical Examples

14.11.1 Dynamic Function Dispatch

```
operations=( [1]="add" [2]="subtract" [3]="multiply" )

add() { echo $(( $1 + $2 )); }
subtract() { echo $(( $1 - $2 )); }
multiply() { echo $(( $1 * $2 )); }

dispatch() {
    local op=$1 a=$2 b=$3
    ${operations[$op]} $a $b
}

dispatch 2 5 3 # Output: 2 (5-3)
```

14.11.2 Concurrent Process Manager

```
max_workers=4
task_queue=()
worker_pids=()

add_task() {
    task_queue+=("$@")
}

start_workers() {
    for ((i=0; i<max_workers; i++)); do
        (
            while true; do
                # Get next task atomically
                flock 200
                if (( ${#task_queue[@]} > 0 )); then
                    task=${task_queue[0]}
                    task_queue="${task_queue[@]:1}"
                else
                    break
                fi
                flock -u 200

                # Execute task
                ${task[@]}
            done
        ) 200> /tmp/worker.lock &
        worker_pids+=($!)
    done
}

wait_workers() {
    wait "${worker_pids[@]}"
}
```

14.11.3 Self-Modifying Script

```
# Modify script behavior at runtime
if [[ -f "config.override" ]]; then
    source "config.override"
    trap 'rm config.override; exec $0 $@' EXIT
fi
```

14.12 Best Practices

- Use `set -euo pipefail` for robust scripts
- Prefer `[[]]` over `[]` for tests
- Always quote variables unless you need word splitting
- Use `local` variables in functions
- Document complex features with comments
- Validate input rigorously
- Consider using `shellcheck` for static analysis

These advanced features enable you to write sophisticated, production-grade Bash scripts that can handle complex tasks efficiently. Would you like me to elaborate on any specific feature?

15 Best Practices in Bash Scripting

Writing robust, maintainable, and secure Bash scripts requires following established best practices. Here's a comprehensive guide to professional Bash scripting:

15.1 Script Structure and Organization

15.1.1 Header Section

```
#!/usr/bin/env bash
#
# Script: myscript.sh
# Purpose: Describe script functionality
# Author: Your Name <your.email@example.com>
# Version: 1.0
# License: MIT
```

15.1.2 Configuration Section

```
# Configuration variables (UPPERCASE by convention)
readonly MAX_RETRIES=3
readonly LOG_FILE="/var/log/myscript.log"
readonly DEFAULT_PORT=8080
```

15.1.3 Function Definitions

```
# Define functions before main execution
validate_input() {
    # Function code here
}
```

15.1.4 Main Execution

```
main() {
    # Main script logic
}

# Entry point
main "$@"
```

15.2 Error Handling

15.2.1 Strict Mode

```
set -euo pipefail # Exit on error, undefined variables, and pipe failures
```

15.2.2 Custom Error Handling

```
trap 'cleanup; exit 1' ERR SIGINT SIGTERM

cleanup() {
    echo "Error occurred. Cleaning up..." >&2
    rm -f "$TEMP_FILE"
}
```

15.2.3 Exit Codes

```
if [ ! -f "$CONFIG_FILE" ]; then
    echo "Config file missing" >&2
    exit 1
fi
```

15.3 Security Practices

15.3.1 Input Validation

```
if [[ ! "$input" =~ ^[a-zA-Z0-9_]+$ ]]; then
    echo "Invalid input" >&2
    exit 1
fi
```

15.3.2 Sanitization

```
# Safely handle filenames
filename=$(basename "$unsafe_input")
```

15.3.3 Privilege Management

```
if [[ $EUID -ne 0 ]]; then
    echo "This script must be run as root" >&2
    exit 1
fi
```

15.4 Coding Standards

15.4.1 Variable Naming

```
local snake_case_variable="value" # Local variables
readonly CONSTANT_VARIABLE="value" # Constants
```

15.4.2 Function Practices

```
# Descriptive function names
calculate_disk_usage() {
    local path=$1 # Document parameters
    # Function body
}
```

15.4.3 Commenting

```
# Check if file exists (comment explains why, not what)
if [ -f "$FILE" ]; then
    # Process valid file
    process_file "$FILE"
fi
```

15.5 Performance Considerations

15.5.1 Subshell Minimization

```
# Instead of:
count=$(ls | wc -l)

# Use:
count=0
for file in *; do
  ((count++))
done
```

15.5.2 Builtin Preference

```
# Use shell builtins when possible
echo "${array[@]}" > file # Instead of printf
```

15.6 Portability

15.6.1 Shebang Portability

```
#!/usr/bin/env bash # Finds bash in PATH
```

15.6.2 Feature Detection

```
# Check for Bash version
if ((BASH_VERSINFO[0] < 4)); then
  echo "Requires Bash 4+" >&2
  exit 1
fi
```

15.6.3 Cross-Platform Paths

```
# Use forward slashes even on Windows (for Cygwin/WSL)
config_path="${HOME}/config/app.cfg"
```

15.7 Documentation

15.7.1 Help Function

```
usage() {
    cat <<EOF
Usage: ${0##*/} [options] <input>

Options:
    -h      Show this help
    -v      Enable verbose mode
    -f FILE Specify input file
EOF
    exit 1
}
```

15.7.2 Inline Documentation

```
: <<'DOC'
This section contains detailed documentation
```

```
about complex algorithm implementation.
DOC
```

15.8 Testing and Debugging

15.8.1 Debug Mode

```
if [[ "$DEBUG" == "true" ]]; then
    set -x # Print commands before execution
fi
```

15.8.2 Linting

```
# Use shellcheck to analyze scripts
# https://www.shellcheck.net/
```

15.8.3 Unit Testing

```
# Simple test framework pattern
test_addition() {
    result=$(add 2 3)
    assert_equals 5 "$result"
}
```

15.9 Maintenance Practices

15.9.1 Version Control

```
# Keep scripts under version control (Git)
# Include .gitignore for temporary files
```

15.9.2 Logging

```
log() {
    local level=$1 message=$2
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $message" >> "$LOG_FILE"
}

log "INFO" "Script started"
```

15.9.3 Dependency Management

```
# Check for required commands
for cmd in jq awk curl; do
    if ! command -v "$cmd" &> /dev/null; then
        echo "Missing required command: $cmd" >&2
        exit 1
    fi
done
```

15.10 Practical Examples

15.10.1 Robust Template

```
#!/usr/bin/env bash
set -euo pipefail
```

```

IFS=$'\n\t'

# Configuration
readonly SCRIPT_NAME="${0##*/}"
readonly VERSION="1.0.0"
readonly LOCK_FILE="/tmp/${SCRIPT_NAME}.lock"

# Functions
acquire_lock() {
    if [ -e "$LOCK_FILE" ]; then
        local pid
        pid=$(cat "$LOCK_FILE")
        if ps -p "$pid" > /dev/null; then
            echo "Script already running (PID: $pid)" >&2
            exit 1
        fi
    fi
    echo $$ > "$LOCK_FILE"
}

release_lock() {
    rm -f "$LOCK_FILE"
}

main() {
    acquire_lock
    trap release_lock EXIT

    # Main script logic here
    echo "Running ${SCRIPT_NAME} version ${VERSION}"
}

# Entry point
main "$@"

```

By following these best practices, you'll create Bash scripts that are:

- More reliable and less prone to errors
- Easier to maintain and modify
- More secure against common vulnerabilities
- Better documented and understood by others
- More portable across different systems

Remember that Bash has limitations, and for very complex tasks, consider using more robust programming languages like Python or Ruby. However, for system automation and glue code, well-written Bash scripts are invaluable.