Linux Shell Scripting for DevOps

Akhilesh Mishra

Contents

1 Linux Shell Scripting For Devops

A Minimalistic Guide to Writing Scripts That Actually Work

A practical, theory-focused guide to mastering shell scripting. This book teaches you essential concepts with simple examples and hands-on exercises.

1.1 About This Book

This book takes a clean, focused approach to teaching shell scripting. Each chapter covers one major topic with: - Clear theoretical explanations - Simple, practical examples - One comprehensive exercise combining chapter concepts

You'll learn to automate tasks, process data, and write production-ready scripts.

1.2 About the Author

Akhilesh Mishra is a DevOps Engineer with expertise in automation, cloud infrastructure, and system administration. With hands-on experience in shell scripting, CI/CD pipelines, and infrastructure as code, he has helped organizations streamline their development and deployment processes.

Throughout his career, Akhilesh has focused on building efficient, scalable systems and sharing knowledge with the technical community through writing, tutorials, and open-source contributions.

Social links

- **Website**: livingdevops.com
- **LinkedIn**: linkedin.com/in/your-profile
- **GitHub**: github.com/akhileshmishrabiz

- **Twitter/X**: @livingdevops
- **Blog**: medium.com/@akhilesh-mishra

1.3 Who This Book Is For

- System administrators
- Developers
- DevOps engineers
- Anyone wanting to master the command line

---

2 Chapter 0: Introduction to Shell Scripting

2.1 What is a Shell Script?

A shell script is a text file containing a sequence of commands for the shell to execute. Think of it as a recipe: you write down the steps once, and the shell follows them automatically every time you run the script.

Without a script:

```
cd /var/log
grep "ERROR" app.log
wc -l
```

With a script:

```
#!/bin/bash
cd /var/log
grep "ERROR" app.log | wc -l
```

You type `./check_errors.sh` instead of remembering and typing multiple commands.

2.2 Why Shell Scripting in DevOps?

DevOps is all about automation, and shell scripts are the glue that holds automation together. Here's why they're essential:

1. Automation - Deploy applications automatically - Backup databases on schedule - Restart services when they fail - Clean up old logs to save space

2. Configuration Management - Setup new servers consistently - Configure applications with environment-specific settings - Manage user accounts and permissions

3. CI/CD Pipelines - Build and test code automatically - Deploy to staging and production - Run health checks after deployment

4. Monitoring and Alerting - Check disk space and send alerts - Monitor application logs for errors - Collect metrics and send to monitoring systems

5. Bridging Tools - Connect Docker, Kubernetes, AWS CLI, and other tools - Orchestrate complex workflows - Handle errors and retry operations

Shell scripts run everywhere: Linux servers, macOS, Docker containers, CI/CD platforms (Jenkins, GitLab CI, GitHub Actions), and cloud instances.

2.3 Your First Shell Script

Let's create a simple script:

Step 1: Create the file

```
touch hello.sh
```

Step 2: Edit the file (use any text editor)

```
#!/bin/bash
# My first shell script

echo "Hello, World!"
echo "Today is $(date)"
echo "I am running on $(hostname)"
```

Step 3: Make it executable

```
chmod +x hello.sh
```

Step 4: Run it

```
./hello.sh
```

Output:

```
Hello, World!
Today is Tue Oct  7 10:30:45 UTC 2025
I am running on server1
```

2.4 Understanding the Shebang (#!/bin/bash)

The first line `#!/bin/bash` is called a **shebang** or **hashbang**. It tells the system which interpreter to use to execute the script.

Anatomy: - `#!` - Special character sequence - `/bin/bash` - Path to the Bash interpreter

Common shebangs:

```
#!/bin/bash              # Use Bash shell
#!/bin/sh                # Use standard shell (more portable)
#!/usr/bin/env bash      # Find Bash in PATH (more portable)
#!/usr/bin/python3       # Python script
```

Why it matters: Without the shebang, you must explicitly specify the interpreter:

```
bash hello.sh           # With shebang: ./hello.sh
```

The shebang makes scripts self-contained and executable like any program.

2.5 Making Scripts Executable

Scripts need execute permission to run directly:

```
# Check current permissions
ls -l hello.sh
# Output: -rw-r--r-- 1 user user 50 Oct 7 10:30 hello.sh

# Add execute permission
chmod +x hello.sh

# Check again
ls -l hello.sh
# Output: -rwxr-xr-x 1 user user 50 Oct 7 10:30 hello.sh
#           ^^ Now executable
```

Permission breakdown: - `r` = read (4) - `w` = write (2) - `x` = execute (1)

Common chmod patterns:

```
chmod +x script.sh       # Add execute for everyone
chmod 755 script.sh      # rwxr-xr-x (owner: all, others: read+execute)
chmod 700 script.sh      # rwx------ (only owner can use)
```

2.6 How to Run Scripts

Method 1: Direct execution (requires execute permission)

```
./script.sh
```

Method 2: Explicit interpreter

```
bash script.sh          # No execute permission needed
```

Method 3: From PATH

```
# Move to directory in PATH
sudo mv script.sh /usr/local/bin/
script.sh               # Run from anywhere
```

2.7 Basic Script Structure

A well-structured script is easier to understand and maintain:

```
#!/bin/bash
# Script: system_info.sh
# Purpose: Display system information
# Author: Your Name

# Exit on error
set -e

# Variables
HOSTNAME=$(hostname)
DATE=$(date)

# Main logic
echo "=== System Information ==="
echo "Hostname: $HOSTNAME"
echo "Date: $DATE"
echo "Uptime: $(uptime -p)"
echo "Disk Usage: $(df -h / | awk 'NR==2 {print $5}')"
```

Key elements: 1. **Shebang** - Which interpreter to use 2. **Comments** - Explain what the script does 3. **Settings** - `set -e` exits on errors 4. **Variables** - Store values for reuse 5. **Logic** - The actual work

2.8 Comments

Comments explain code and are ignored by the shell:

```
# This is a single-line comment

echo "Hello"  # Comment after code

# Multi-line comments:
: '
This is a multi-line comment.
Everything between the quotes is ignored.
Useful for documentation.
'
```

When to comment: - Explain *why*, not *what* (code shows what) - Document complex logic - Provide usage examples - Add TODO notes

2.9 Variables Basics

Variables store data for later use:

```
# Assign (no spaces around =)
name="DevOps"
count=42

# Use (prefix with $)
echo "Hello, $name"
echo "Count: $count"

# Command output
current_user=$(whoami)
file_count=$(ls | wc -l)
```

Rules: - No spaces around `=` - Quote values with spaces: `name="John Doe"` - Use `$variable` or `${variable}` to access

2.10 Testing Your Scripts

Always test scripts before deploying:

1. Syntax check

```
bash -n script.sh       # Check for syntax errors
```

2. Dry run with debug

```
bash -x script.sh       # Show each command before execution
```

3. Test with various inputs

```
./script.sh test1.txt
./script.sh /path/to/file
./script.sh ""          # Empty input
```

4. Check exit codes

```
./script.sh
echo $?                 # 0 = success, non-zero = error
```

2.11 Common Beginner Mistakes

1. Forgetting the shebang

```
# Wrong
echo "hello"

# Correct
#!/bin/bash
echo "hello"
```

2. Spaces around = in assignments

```
# Wrong
name = "value"

# Correct
name="value"
```

3. Not quoting variables

```
# Wrong
echo $file_name

# Correct
echo "$file_name"
```

4. Forgetting execute permission

```
# Wrong
script.sh               # Permission denied

# Correct
chmod +x script.sh
./script.sh
```

5. Wrong path to script

```
# Wrong
script.sh               # Command not found

# Correct
./script.sh             # Current directory
```

2.12 Exercise: System Information Script

Create a script that displays useful system information:

```
#!/bin/bash
# system_info.sh - Display system information

echo "=================================="
echo "    System Information Report"
echo "=================================="
echo ""

echo "Date and Time:"
echo "  $(date)"
echo ""

echo "System:"
echo "  Hostname: $(hostname)"
echo "  OS: $(uname -s)"
echo "  Kernel: $(uname -r)"
echo ""

echo "User:"
echo "  Current user: $(whoami)"
```

```
echo "  Home directory: $HOME"
echo "  Shell: $SHELL"
echo ""

echo "System Load:"
echo "  Uptime: $(uptime -p 2>/dev/null || uptime)"
echo ""

echo "Disk Usage:"
df -h / | awk 'NR==1 {print "  " $0} NR==2 {print "  " $0}'
echo ""

echo "Memory:"
free -h 2>/dev/null | awk 'NR==1 {print "  " $0} NR==2 {print "  " $0}' || echo "  (memory inf
echo ""

echo "==================================="
echo "Report generated by: $0"
echo "==================================="
```

Create and run:

```
# Create file
cat > system_info.sh << 'EOF'
[paste script above]
EOF

# Make executable
chmod +x system_info.sh

# Run it
./system_info.sh
```

This script demonstrates: - Proper structure with shebang and comments - Using variables (`$HOME`, `$SHELL`) - Command substitution (`$(date)`, `$(hostname)`) - Output formatting - Error handling (`2>/dev/null`)

2.13 Summary

Shell scripting is fundamental to DevOps automation. Scripts combine multiple commands into repeatable processes, essential for deployment, monitoring, and system management. Every script needs a shebang (`#!/bin/bash`), execute permission (`chmod +x`), and proper structure with comments and variables.

Start with simple scripts and gradually build complexity as you learn more concepts in the following chapters.

---

3 Chapter 1: Environment Variables

3.1 Why This Matters in DevOps

In DevOps workflows, environment variables are crucial for:

Configuration Management: Different environments (dev, staging, production) need different settings. Instead of hardcoding database URLs or API keys, use environment variables that change based on where the code runs.

12-Factor App Methodology: Modern cloud applications store configuration in environment variables, making apps portable across environments without code changes.

CI/CD Pipelines: Jenkins, GitLab CI, GitHub Actions all use environment variables to pass secrets, build settings, and deployment parameters to your scripts.

Container Orchestration: Docker and Kubernetes inject configuration through environment variables, allowing the same container image to run with different configurations.

Security: Sensitive data (API keys, passwords) shouldn't be in code. Environment variables keep secrets out of version control while making them available at runtime.

Example: A deployment script uses `$DATABASE_URL`, `$API_KEY`, and `$ENVIRONMENT` to connect to the right database with correct credentials based on where it's running.

Environment variables are named values that store configuration data. The operating system and programs use them to control behavior without hardcoding values.

3.2 Why Environment Variables Matter

Think of environment variables as global settings for your system. They tell programs: - Where to find commands (PATH) - Where your home directory is (HOME) - Which editor to use (EDITOR)

- What language to display (LANG)

This separation of configuration from code makes systems flexible and portable.

3.3 Reading Environment Variables

View a specific variable:

```
echo $HOME
echo $USER
echo $PATH
```

The `$` tells the shell to substitute the variable's value. `$HOME` becomes `/home/username`.

View all variables:

```
env                    # All environment variables
printenv HOME          # Specific variable
```

3.4 Setting Environment Variables

For current shell only:

```
MY_VAR="hello"
echo $MY_VAR
```

Export to child processes:

```
export MY_VAR="hello"
./my_script.sh         # Script can access MY_VAR
```

Without `export`, child processes don't see the variable. Use `export` when you want programs you run to access the variable.

Temporary variable for one command:

```
DEBUG=true ./script.sh
```

This sets `DEBUG` only for that one command execution.

3.5 The PATH Variable

`PATH` is the most important environment variable. It lists directories where the shell looks for commands.

```
echo $PATH
# Output: /usr/bin:/bin:/usr/local/bin
```

```
which ls                # Shows: /usr/bin/ls
```

When you type `ls`, the shell searches each PATH directory until it finds the `ls` program.

Add to PATH:

```
export PATH="$PATH:/new/directory"
```

Always include `$PATH` first to preserve existing paths.

3.6 Default Values

Use `${VAR:-default}` to provide fallback values:

```
LOG_DIR="${LOG_DIR:-/var/log}"
DEBUG="${DEBUG:-false}"
```

If `LOG_DIR` isn't set, it defaults to `/var/log`. This makes scripts configurable without requiring every variable to be set.

3.7 Exercise: Environment Configuration Script

```
#!/bin/bash
# Show system environment and create custom project environment

echo "=== System Environment ==="
echo "User: $USER"
echo "Home: $HOME"
echo "Shell: $SHELL"
echo ""

# Count PATH directories
IFS=: read -ra paths <<< "$PATH"
echo "PATH has ${#paths[@]} directories"

# Create project environment with defaults
export PROJECT_NAME="${PROJECT_NAME:-MyApp}"
export PROJECT_ENV="${PROJECT_ENV:-development}"
export PROJECT_PORT="${PROJECT_PORT:-8080}"

echo ""
```

```
echo "=== Project Environment ==="
echo "Name: $PROJECT_NAME"
echo "Environment: $PROJECT_ENV"
echo "Port: $PROJECT_PORT"

# Save to file
cat > .env <<EOF
PROJECT_NAME=$PROJECT_NAME
PROJECT_ENV=$PROJECT_ENV
PROJECT_PORT=$PROJECT_PORT
EOF

echo ""
echo "Configuration saved to .env"
```

Run it:

```
chmod +x env_config.sh
./env_config.sh

# Or with custom values:
PROJECT_NAME="MyAPI" PROJECT_PORT=3000 ./env_config.sh
```

---

4 Chapter 2: Command Line Arguments

4.1 Why This Matters in DevOps

Command line arguments make scripts flexible and reusable across different scenarios:

Deployment Scripts: Pass environment name, version number, or target servers as arguments instead of editing the script each time.

Backup Automation: Specify what to backup, where to store it, and retention period through arguments, making one script handle multiple backup strategies.

Infrastructure Provisioning: Pass instance types, regions, and configurations as parameters when creating cloud resources.

CI/CD Integration: Build pipelines pass build numbers, branch names, and deployment targets as arguments to your scripts.

Example: `./deploy.sh production v2.5.0 us-east-1` - one script handles all deployments by accepting different parameters.

Command line arguments make scripts reusable by accepting input when you run them. Instead of hardcoding values, you pass them as parameters.

4.2 How Arguments Work

When you run a script with arguments:

```
./script.sh alice 25 london
```

The shell assigns them to special variables: - `$0` = script name (`./script.sh`) - `$1` = first argument (`alice`) - `$2` = second argument (`25`) - `$3` = third argument (`london`) - `$#` = count of arguments (`3`) - `$@` = all arguments as separate words

4.3 Basic Argument Usage

```
#!/bin/bash
# greet.sh - Simple greeting script

name=$1
age=$2

echo "Hello, $name!"
echo "You are $age years old."
```

Run it:

```
chmod +x greet.sh
./greet.sh Alice 25
```

4.4 Providing Default Values

```
#!/bin/bash
name=${1:-"Guest"}
age=${2:-"unknown"}

echo "Hello, $name!"
echo "You are $age years old."
```

The `${1:-"Guest"}` syntax means: use `$1` if provided, otherwise use "Guest".

4.5 Validating Arguments

Always check if required arguments are provided:

```
#!/bin/bash
if [ $# -eq 0 ]; then
    echo "Usage: $0 <name> [age]"
    exit 1
fi

name=$1
age=${2:-"unknown"}

echo "Hello, $name! Age: $age"
```

`$#` counts arguments. If zero, show usage and exit with error code 1.

4.6 Exit Status

Scripts return an exit code: - `0` = success - `1` = error

Check with `$?`:

```
./script.sh
echo $?              # Shows exit code
```

4.7 Handling Spaces

Arguments with spaces need quotes:

```
./script.sh "John Doe" 30
```

Without quotes, "John" becomes `$1` and "Doe" becomes `$2`.

4.8 Exercise: File Information Script

```
#!/bin/bash
# fileinfo.sh - Display file information with validation

# Validate argument
if [ $# -eq 0 ]; then
    echo "Usage: $0 <filename>"
    exit 1
```

```
fi

file=$1

# Check if file exists
if [ ! -f "$file" ]; then
    echo "Error: File '$file' not found"
    exit 2
fi

# Display information
echo "=== File Information ==="
echo "Name: $(basename "$file")"
echo "Path: $(realpath "$file")"
echo "Size: $(wc -c < "$file") bytes"
echo "Lines: $(wc -l < "$file")"
echo "Words: $(wc -w < "$file")"

# Check permissions
echo ""
echo "=== Permissions ==="
[ -r "$file" ] && echo "  Readable" || echo "  Not readable"
[ -w "$file" ] && echo "  Writable" || echo "  Not writable"
[ -x "$file" ] && echo "  Executable" || echo "  Not executable"

echo ""
echo "Last modified: $(date -r "$file")"
```

Test it:

```
# Create test file
echo "Hello World" > test.txt

# Run script
chmod +x fileinfo.sh
./fileinfo.sh test.txt
./fileinfo.sh                    # Should show usage
./fileinfo.sh nonexistent.txt    # Should show error
```

5 Chapter 3: Wildcards and Pattern Matching

5.1 Why This Matters in DevOps

Wildcards are essential for batch operations on files:

Log Management: Archive all logs from yesterday (`*.log.2025-10-06`), compress old logs (`*.log.*`), or delete debug files (`*debug*.txt`).

Deployment Cleanup: Remove old builds (`build-*.tar.gz`), clean up temporary files (`tmp*`), or backup configuration files (`*.conf`).

Monitoring Scripts: Process all service logs (`/var/log/service*.log`), check all configuration files in a directory, or scan multiple log sources.

Batch Processing: Convert all images (`*.jpg`), process all CSV files, or validate all configuration files matching a pattern.

Example: `tar -czf logs-backup.tar.gz *.log` - backup all log files with one command instead of listing each file.

Wildcards let you match multiple files with patterns instead of typing each filename. They're essential for batch operations.

5.2 Why Wildcards Matter

Instead of:

```
rm file1.txt file2.txt file3.txt file4.txt
```

You write:

```
rm *.txt
```

Wildcards make scripts work with any number of files without modification.

5.3 The Three Main Wildcards

1. Star (*) - matches any characters

```
*.txt           # All files ending in .txt
report*         # All files starting with report
*error*         # All files containing error
```

2. Question mark (?) - matches exactly one character

```
file?.txt       # file1.txt, file2.txt (not file10.txt)
???.log         # Any 3-character name with .log
```

3. Square brackets [] - matches specific characters

```
file[123].txt   # file1.txt, file2.txt, file3.txt
[A-Z]*          # Files starting with uppercase letter
[!0-9]*         # Files NOT starting with digit
```

5.4 Brace Expansion

Create multiple strings at once:

```
{1..5}          # Expands to: 1 2 3 4 5
file{1..3}.txt  # Creates: file1.txt file2.txt file3.txt
*.{jpg,png}     # Matches: all .jpg and .png files
```

5.5 Practical Examples

Setup test files:

```
touch report{1..5}.txt
touch data{1..3}.csv
touch log_{a,b,c}.txt
```

Use wildcards:

```
ls *.txt                    # All text files
ls report[1-3].txt          # report1, report2, report3
ls log_?.txt                # log_a, log_b, log_c
rm *.csv                    # Delete all CSV files
```

5.6 Safety First

Always test with `ls` before using destructive commands:

```
ls *.txt        # Verify what matches
rm *.txt        # Then delete if correct
```

5.7 Exercise: File Organizer

```
#!/bin/bash
# organize.sh - Organize files by type into directories
```

```
echo "=== File Organizer ==="

# Create test files
echo "Creating test files..."
touch document{1..3}.txt
touch photo{1..3}.jpg
touch data{1..3}.csv
touch script{1..3}.sh

# Create directories
mkdir -p documents images data scripts

# Count and move files
txt_count=$(ls *.txt 2>/dev/null | wc -l)
jpg_count=$(ls *.jpg 2>/dev/null | wc -l)
csv_count=$(ls *.csv 2>/dev/null | wc -l)
sh_count=$(ls *.sh 2>/dev/null | wc -l)

echo ""
echo "Found files:"
echo "  Text files: $txt_count"
echo "  Images: $jpg_count"
echo "  CSV files: $csv_count"
echo "  Scripts: $sh_count"

# Move files
[ $txt_count -gt 0 ] && mv *.txt documents/
[ $jpg_count -gt 0 ] && mv *.jpg images/
[ $csv_count -gt 0 ] && mv *.csv data/
[ $sh_count -gt 0 ] && mv *.sh scripts/

echo ""
echo "Files organized:"
ls -R documents/ images/ data/ scripts/

# Cleanup
```

```
echo ""
read -p "Remove organized directories? (y/n) " answer
if [ "$answer" = "y" ]; then
    rm -rf documents/ images/ data/ scripts/
    echo "Cleaned up."
fi
```

---

6 Chapter 4: String Manipulation

6.1 Why This Matters in DevOps

String manipulation is crucial for parsing and transforming data:

Path Processing: Extract filenames from full paths, remove extensions, or build new paths dynamically in deployment scripts.

Configuration Parsing: Extract values from config files, transform environment-specific settings, or build connection strings.

Version Management: Parse semantic version strings (`v2.5.3`), compare versions, or extract major/minor/patch numbers.

Log Analysis: Extract timestamps, parse structured log entries, or filter specific patterns from log messages.

Dynamic Resource Naming: Build AWS resource names, Docker tags, or Kubernetes labels by manipulating strings based on environment, branch, or version.

Example: Extract deployment version from git tag `release/v2.5.3` → extract `2.5.3` for Docker image tagging.

Bash provides built-in operations to extract, remove, and replace text without external commands. These operations are fast and don't require tools like sed or awk.

6.2 Why String Manipulation Matters

Scripts often need to: - Extract filenames from paths - Remove file extensions - Replace text in variables - Parse configuration values

Built-in string operations are faster than spawning external processes.

6.3 String Length

```
text="Hello World"
echo ${#text}              # 11
```

The # inside ${} means length.

6.4 Extracting Substrings

```
text="Hello World"
echo ${text:0:5}           # Hello (start at 0, length 5)
echo ${text:6}             # World (start at 6, to end)
echo ${text: -5}           # World (last 5 characters, note the space)
```

6.5 Removing from Beginning

```
file="/home/user/report.txt"
echo ${file#*/}            # home/user/report.txt (remove shortest match)
echo ${file##*/}           # report.txt (remove longest match)
```

- # removes shortest matching pattern from beginning
- ## removes longest matching pattern from beginning

6.6 Removing from End

```
file="/home/user/report.txt"
echo ${file%.*}            # /home/user/report (remove extension)
echo ${file%%/*}           # (empty, removes everything after first /)
```

- % removes shortest matching pattern from end
- %% removes longest matching pattern from end

6.7 Pattern Replacement

```
text="hello world hello"
echo ${text/hello/hi}      # hi world hello (first occurrence)
echo ${text//hello/hi}     # hi world hi (all occurrences)
```

6.8 Case Conversion

```
text="Hello World"
echo ${text^^}             # HELLO WORLD (uppercase)
echo ${text,,}             # hello world (lowercase)
```

6.9 Exercise: Path and Filename Parser

```
#!/bin/bash
# pathparser.sh - Parse file paths and manipulate strings

# Test file path
path="/home/user/documents/report_2024.txt"

echo "=== Original Path ==="
echo "$path"
echo ""

# Extract components
filename="${path##*/}"              # Get filename only
dirname="${path%/*}"                # Get directory only
basename="${filename%.*}"           # Filename without extension
extension="${filename##*.}"         # Extension only

echo "=== Components ==="
echo "Directory: $dirname"
echo "Filename: $filename"
echo "Base name: $basename"
echo "Extension: $extension"
echo ""

# Transform filename
uppercase="${filename^^}"
lowercase="${filename,,}"
renamed="${filename/report/summary}"

echo "=== Transformations ==="
echo "Uppercase: $uppercase"
echo "Lowercase: $lowercase"
echo "Renamed: $renamed"
echo ""

# Build new path
```

```
new_path="${dirname}/${renamed}"
echo "New path: $new_path"
echo ""

# Extract year from filename
if [[ $filename =~ ([0-9]{4}) ]]; then
    year="${BASH_REMATCH[1]}"
    echo "Found year: $year"
fi
```

---

7 Chapter 5: Arithmetic Operations

7.1 Why This Matters in DevOps

Arithmetic operations enable calculations and threshold checking:

Resource Monitoring: Calculate disk usage percentages, memory consumption, or CPU thresholds to trigger alerts.

Capacity Planning: Compute required resources, calculate costs, or determine if autoscaling is needed based on current usage.

Retry Logic: Increment counters, calculate exponential backoff delays, or track failed attempts.

Performance Metrics: Calculate averages, track response times, or compute throughput rates.

Batch Processing: Count processed items, calculate progress percentages, or split work across multiple workers.

Example: `if (( disk_usage > 80 )); then send_alert; fi` - trigger alerts when disk usage exceeds threshold.

Bash performs integer arithmetic natively using `$(( ))` syntax. For decimal math, use the `bc` calculator.

7.2 Integer Arithmetic

Use double parentheses for math:

```
echo $((5 + 3))           # 8
echo $((10 - 4))          # 6
echo $((6 * 7))           # 42
```

```
echo $((20 / 4))            # 5
echo $((17 % 5))            # 2 (remainder)
echo $((2 ** 8))            # 256 (power)
```

Variables don't need `$` inside `$(( ))`:

```
x=10
y=3
echo $((x + y))             # 13
```

7.3 Assignment Operators

```
count=5
((count++))                 # count = 6
((count += 10))             # count = 16
((count *= 2))              # count = 32
```

7.4 Arithmetic in Conditions

```
x=10
if ((x > 5)); then
    echo "x is greater than 5"
fi

for ((i=0; i<5; i++)); do
    echo $i
done
```

7.5 Decimal Math with bc

Bash only does integer division:

```
echo $((10 / 3))            # 3 (integer division)
```

Use `bc` for decimals:

```
echo "scale=2; 10/3" | bc   # 3.33

# With variables
x=10
y=3
```

```
result=$(echo "scale=2; $x / $y" | bc)
echo $result                # 3.33
```

The `scale` parameter sets decimal places.

7.6 Random Numbers

```
echo $RANDOM                # Random number 0-32767
echo $((RANDOM % 100))      # Random 0-99
echo $((RANDOM % 100 + 1)) # Random 1-100
```

7.7 Exercise: Calculator Script

```
#!/bin/bash
# calc.sh - Simple calculator with integer and decimal support

echo "=== Simple Calculator ==="
echo ""

# Get input
read -p "Enter first number: " num1
read -p "Enter operator (+, -, *, /): " op
read -p "Enter second number: " num2

# Integer calculation
case $op in
    +) result=$((num1 + num2));;
    -) result=$((num1 - num2));;
    *) result=$((num1 * num2));;
    /) result=$((num1 / num2));;
    *) echo "Invalid operator"; exit 1;;
esac

echo ""
echo "=== Integer Result ==="
echo "$num1 $op $num2 = $result"

# Decimal calculation using bc
if [ "$op" = "/" ]; then
```

```
    decimal=$(echo "scale=2; $num1 / $num2" | bc)
    echo ""
    echo "=== Decimal Result ==="
    echo "$num1 $op $num2 = $decimal"
fi

# Additional calculations
echo ""
echo "=== Additional Info ==="
echo "Sum: $((num1 + num2))"
echo "Difference: $((num1 - num2))"
echo "Product: $((num1 * num2))"

if [ $num2 -ne 0 ]; then
    echo "Quotient: $((num1 / num2))"
    echo "Remainder: $((num1 % num2))"
fi
```

---

8 Chapter 6: Conditional Expressions

8.1 Why This Matters in DevOps

Conditionals make scripts intelligent and defensive:

Pre-flight Checks: Verify files exist before processing, check if services are running, or validate prerequisites before deployment.

Environment Detection: Execute different logic for dev vs production, handle OS differences, or adapt to available tools.

Error Prevention: Check disk space before backups, validate configuration before applying, or verify connectivity before operations.

Smart Automation: Only restart services if they're down, skip backups if nothing changed, or deploy only if tests pass.

Security Validation: Verify file permissions, check user privileges, or validate input before execution.

Example: `if [[ -f "config.yml" && -r "config.yml" ]]; then load_config; else error`

`"Config not found"; fi` - robust configuration loading.

Conditional expressions test files, strings, and numbers to make decisions in scripts. They control program flow based on conditions.

8.2 Why Conditionals Matter

Scripts need to make decisions: - Does a file exist before reading it? - Is a variable empty? - Is a number within range?

Conditionals prevent errors and enable intelligent behavior.

8.3 The Modern Test Syntax

Always use `[[ ]]` (double brackets):

```
[[ -f file.txt ]]          # Good - modern, safe
[ -f file.txt ]            # Old - has limitations
test -f file.txt           # Oldest - avoid
```

Double brackets handle spaces in variables correctly and support more features.

8.4 File Tests

```
[[ -f file ]]              # Is a regular file
[[ -d dir ]]               # Is a directory
[[ -e path ]]              # Exists (any type)
[[ -r file ]]              # Is readable
[[ -w file ]]              # Is writable
[[ -x file ]]              # Is executable
[[ -s file ]]              # Exists and not empty
```

8.5 String Tests

```
[[ -z "$str" ]]            # Is empty (zero length)
[[ -n "$str" ]]            # Is not empty
[[ "$a" == "$b" ]]         # Strings equal
[[ "$a" != "$b" ]]         # Strings not equal
[[ "$str" == *.txt ]]      # Pattern matching
```

8.6 Numeric Comparisons

```
[[ $x -eq 5 ]]              # Equal
[[ $x -ne 5 ]]              # Not equal
[[ $x -lt 5 ]]              # Less than
[[ $x -le 5 ]]              # Less than or equal
[[ $x -gt 5 ]]              # Greater than
[[ $x -ge 5 ]]              # Greater than or equal

# Or use arithmetic comparison:
((x > 5))                   # Cleaner for numbers
```

8.7 Combining Conditions

```
# AND
[[ -f "$file" && -r "$file" ]]

# OR
[[ "$user" == "admin" || "$user" == "root" ]]

# NOT
[[ ! -e "$file" ]]

# Complex
[[ (-f "$file" || -L "$file") && -r "$file" ]]
```

8.8 Regular Expressions

```
email="user@example.com"
if [[ $email =~ ^[^@]+@[^@]+$ ]]; then
    echo "Valid email format"
fi
```

The `=~` operator matches regex patterns.

8.9 Exercise: File Validator

```
#!/bin/bash
# validate.sh - Validate files with multiple checks
```

```
echo "=== File Validator ==="
echo ""

# Get filename
read -p "Enter filename: " file

# Check existence
if [[ ! -e "$file" ]]; then
    echo "  File does not exist"
    exit 1
fi
echo "  File exists"

# Check type
if [[ -f "$file" ]]; then
    echo "  Regular file"
elif [[ -d "$file" ]]; then
    echo "! Directory (not a file)"
    exit 1
else
    echo "! Special file type"
fi

# Check permissions
echo ""
echo "Permissions:"
[[ -r "$file" ]] && echo "  Readable" || echo "  Not readable"
[[ -w "$file" ]] && echo "  Writable" || echo "  Not writable"
[[ -x "$file" ]] && echo "  Executable" || echo "  Not executable"

# Check if empty
echo ""
if [[ -s "$file" ]]; then
    size=$(wc -c < "$file")
    echo "  File has content ($size bytes)"
else
    echo "! File is empty"
```

```
fi

# Check extension
echo ""
if [[ "$file" == *.txt ]]; then
    echo "Text file detected"
    lines=$(wc -l < "$file")
    echo "Lines: $lines"
elif [[ "$file" == *.sh ]]; then
    echo "Shell script detected"
    [[ -x "$file" ]] || echo "! Warning: Script not executable"
fi

echo ""
echo "Validation complete"
```

---

9 Chapter 7: Loops

9.1 Why This Matters in DevOps

Loops automate repetitive operations across multiple targets:

Multi-Server Operations: Deploy to multiple servers, restart services across a cluster, or collect logs from all instances.

Batch Processing: Process all files in a directory, convert multiple formats, or validate all configuration files.

Health Checks: Continuously monitor services, wait for application startup, or poll for job completion.

Resource Iteration: Loop through AWS regions, process all Docker containers, or handle multiple database backups.

Retry Mechanisms: Keep retrying failed operations, wait for resources to become available, or implement exponential backoff.

Example: `for server in web1 web2 web3; do ssh "$server" 'systemctl restart nginx'; done` - restart nginx on all web servers.

Loops execute commands repeatedly. Use `for` loops for known lists, `while` loops for conditions, and `until` loops for waiting.

9.2 For Loops

Process each item in a list:

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

With ranges:

```
for i in {1..5}; do
    echo "Number $i"
done
```

C-style:

```
for ((i=0; i<5; i++)); do
    echo $i
done
```

9.3 While Loops

Continue while condition is true:

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

Infinite loop:

```
while true; do
    echo "Running..."
    sleep 1
done
```

9.4 Until Loops

Continue until condition becomes true (opposite of while):

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

9.5 Reading Files

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

The `-r` prevents backslash interpretation.

9.6 Loop Control

```
for i in {1..10}; do
    [ $i -eq 5 ] && continue    # Skip 5
    [ $i -eq 8 ] && break        # Stop at 8
    echo $i
done
```

- `continue` skips to next iteration
- `break` exits loop completely

9.7 Exercise: System Monitor

```
#!/bin/bash
# monitor.sh - Simple system monitoring with loops

echo "=== System Monitor ==="
echo ""

# Configuration
MAX_CHECKS=5
INTERVAL=2

echo "Will check system $MAX_CHECKS times, every $INTERVAL seconds"
echo "Press Ctrl+C to stop early"
echo ""
```

```
# Monitor loop
for ((i=1; i<=MAX_CHECKS; i++)); do
    echo "=== Check $i of $MAX_CHECKS at $(date +%H:%M:%S) ==="

    # CPU load
    load=$(uptime | awk -F'load average:' '{print $2}' | awk '{print $1}')
    echo "CPU Load: $load"

    # Memory
    mem_used=$(free -h | awk '/Mem:/ {print $3}')
    mem_total=$(free -h | awk '/Mem:/ {print $2}')
    echo "Memory: $mem_used / $mem_total"

    # Disk
    disk=$(df -h / | awk 'NR==2 {print $5}')
    echo "Disk Usage: $disk"

    echo ""

    # Wait before next check (unless last iteration)
    [ $i -lt $MAX_CHECKS ] && sleep $INTERVAL
done

echo "Monitoring complete"
```

Test it:

```
chmod +x monitor.sh
./monitor.sh
```

---

10 Chapter 8: Functions

10.1 Why This Matters in DevOps

Functions create reusable, maintainable automation code:

Code Reusability: Write logging functions once, use everywhere. Create utility functions for

common tasks like checking service status or validating inputs.

Script Libraries: Build shared function libraries that multiple scripts can source - standard logging, error handling, or AWS operations.

Cleaner Scripts: Break complex deployments into logical functions - `validate_prerequisites()`, `backup_current()`, `deploy_new()`, `verify_deployment()`.

Testing: Test individual functions independently, making debugging easier and scripts more reliable.

Team Standards: Establish team-wide function libraries for consistent error handling, logging formats, and operation patterns.

Example: Define `log_info()`, `log_error()`, `send_slack_alert()` once, use throughout all your automation scripts.

Functions group related commands into reusable blocks. They make scripts modular, easier to read, and maintain.

10.2 Why Functions Matter

Without functions:

```
echo "Processing file1.txt"
wc -l file1.txt
grep error file1.txt

echo "Processing file2.txt"
wc -l file2.txt
grep error file2.txt
```

With functions:

```
process_file() {
    echo "Processing $1"
    wc -l "$1"
    grep error "$1"
}

process_file file1.txt
process_file file2.txt
```

Functions eliminate repetition and centralize logic.

10.3 Basic Function Syntax

```
# Define
function_name() {
    commands
}

# Call
function_name
```

10.4 Function Parameters

Functions receive arguments like scripts:

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

greet Alice              # Hello, Alice!
greet Bob                # Hello, Bob!
```

- `$1`, `$2`, etc. = arguments
- `$#` = argument count
- `$@` = all arguments

10.5 Local Variables

Always use `local` to avoid conflicts:

```
my_function() {
    local temp="value"    # Only exists in function
    echo $temp
}

my_function
echo $temp                # Empty - temp doesn't exist here
```

Without `local`, variables affect the whole script.

10.6 Return Values

Functions return exit status (0-255):

```
check_file() {
    [ -f "$1" ] && return 0 || return 1
}

if check_file "test.txt"; then
    echo "File exists"
fi
```

To return data, use `echo`:

```
get_date() {
    echo $(date +%Y-%m-%d)
}

today=$(get_date)
echo $today
```

10.7 Exercise: Log Utility Library

```
#!/bin/bash
# logger.sh - Reusable logging functions

# Configuration
LOG_FILE="app.log"

# Initialize log file
init_log() {
    echo "=== Log started at $(date) ===" > "$LOG_FILE"
}

# Log with timestamp
log() {
    local level=$1
    shift
    local message="$@"
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')

    echo "[$timestamp] [$level] $message" | tee -a "$LOG_FILE"
}
```

```
# Convenience functions
log_info() {
    log "INFO" "$@"
}

log_error() {
    log "ERROR" "$@"
}

log_warn() {
    log "WARN" "$@"
}

# Validate file function
validate_file() {
    local file=$1

    if [[ ! -f "$file" ]]; then
        log_error "File not found: $file"
        return 1
    fi

    if [[ ! -r "$file" ]]; then
        log_error "File not readable: $file"
        return 2
    fi

    log_info "File validated: $file"
    return 0
}

# Main demo
main() {
    init_log

    log_info "Application started"
```

```
    log_info "Processing files..."

    # Create test file
    echo "test" > test.txt

    # Validate file
    if validate_file "test.txt"; then
        log_info "Processing test.txt"
        lines=$(wc -l < test.txt)
        log_info "File has $lines lines"
    fi

    # Test error
    validate_file "nonexistent.txt"

    log_warn "This is a warning"
    log_info "Application finished"

    echo ""
    echo "=== Log Contents ==="
    cat "$LOG_FILE"

    # Cleanup
    rm -f test.txt "$LOG_FILE"
}

main
```

---

11 Chapter 9: Arrays

11.1 Why This Matters in DevOps

Arrays organize and process collections of data:

Server Lists: Manage lists of servers, IP addresses, or hostnames for deployment or monitoring operations.

Configuration Management: Store multiple configuration values, environment settings, or feature flags.

Batch Operations: Collect files to process, services to restart, or containers to manage.

Status Tracking: Track job states, deployment results across multiple targets, or health check outcomes.

Dynamic Infrastructure: Build resource lists from cloud APIs, process multiple environments, or handle variable-length data.

Example: `servers=(web1 web2 web3); for s in "${servers[@]}"; do deploy $s; done` - deploy to all servers using array iteration.

Arrays store multiple values in a single variable. Use indexed arrays for lists and associative arrays for key-value pairs.

11.2 Why Arrays Matter

Without arrays:

```
server1="web1.example.com"
server2="web2.example.com"
server3="web3.example.com"
```

With arrays:

```
servers=("web1.example.com" "web2.example.com" "web3.example.com")
```

Arrays group related data and make it easy to iterate.

11.3 Indexed Arrays

```
# Create
fruits=("apple" "banana" "orange")

# Access
echo ${fruits[0]}           # apple
echo ${fruits[1]}           # banana
echo ${fruits[@]}           # all elements
echo ${#fruits[@]}          # length (3)

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

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

Always quote `"${array[@]}"` to preserve spaces in elements.

11.4 Associative Arrays

Key-value pairs (requires Bash 4+):

```bash
# Declare
declare -A person

# Set
person[name]="John"
person[age]=30
person[city]="NYC"

# Access
echo ${person[name]}        # John
echo ${!person[@]}          # all keys
echo ${person[@]}           # all values
```

11.5 Array Operations

```bash
# Length
echo ${#array[@]}

# Slice
echo ${array[@]:1:2}        # 2 elements starting at index 1

# Loop with indices
for i in "${!array[@]}"; do
    echo "Index $i: ${array[$i]}"
done
```

11.6 Exercise: Task Manager

```
#!/bin/bash
# tasks.sh - Simple task manager using arrays

# Task list
declare -a tasks=()
declare -A task_status=()

# Add task
add_task() {
    local task="$1"
    tasks+=("$task")
    task_status["$task"]="pending"
    echo "  Added: $task"
}

# List tasks
list_tasks() {
    echo "=== Tasks ==="

    if [ ${#tasks[@]} -eq 0 ]; then
        echo "No tasks"
        return
    fi

    for i in "${!tasks[@]}"; do
        local task="${tasks[$i]}"
        local status="${task_status[$task]}"
        printf "%d. [%s] %s\n" $((i+1)) "$status" "$task"
    done
}

# Complete task
complete_task() {
    local index=$1
    local task="${tasks[$index]}"
```

```
    task_status["$task"]="done"
    echo "  Completed: $task"
}

# Demo usage
echo "=== Task Manager Demo ==="
echo ""

add_task "Write documentation"
add_task "Review code"
add_task "Deploy application"
add_task "Update tests"

echo ""
list_tasks

echo ""
echo "Completing task #2..."
complete_task 1

echo ""
list_tasks

# Summary
echo ""
echo "=== Summary ==="
pending=0
done=0

for task in "${tasks[@]}"; do
    [ "${task_status[$task]}" = "pending" ] && ((pending++))
    [ "${task_status[$task]}" = "done" ] && ((done++))
done

echo "Total: ${#tasks[@]}"
echo "Pending: $pending"
echo "Done: $done"
```

12 Chapter 10: Command Substitution

12.1 Why This Matters in DevOps

Command substitution captures dynamic data for use in automation:

Dynamic Configuration: Capture current timestamp for log files, get hostname for identification, or fetch instance metadata from cloud APIs.

Status Checks: Get service status, capture health check results, or determine current deployment version.

Resource Discovery: Find running containers, list available servers, or query infrastructure state.

Calculation and Metrics: Count active connections, measure response times, or calculate resource usage for decision making.

Integration: Combine outputs from multiple tools - get git commit hash for Docker tags, fetch AWS instance IDs, or parse kubectl output.

Example: `COMMIT=$(git rev-parse --short HEAD); docker build -t "app:$COMMIT" .` - tag Docker images with git commit hash.

Command substitution captures command output into variables or uses it as arguments. It's essential for using one command's output as another's input.

12.2 Basic Syntax

Use `$()` (not backticks):

```
today=$(date)
files=$(ls)
user=$(whoami)

echo "Today is $today"
echo "Current user: $user"
```

Why `$()` instead of backticks: - Nests easily: `$(dirname $(pwd))` - More readable - Easier to escape

12.3 Common Uses

Capture command output:

```
file_count=$(ls | wc -l)
disk_usage=$(df -h / | awk 'NR==2 {print $5}')
current_dir=$(pwd)
```

Use in conditions:

```
if [ "$(whoami)" = "root" ]; then
    echo "Running as root"
fi
```

Use in loops:

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

12.4 Process Substitution

Treat command output as a file:

```
# Compare outputs
diff <(ls dir1) <(ls dir2)

# Sort and compare
comm <(sort file1) <(sort file2)
```

`<(command)` creates a temporary file descriptor containing the command's output.

12.5 Avoiding Subshells

Problem:

```
# Variable lost - pipe creates subshell
total=0
ls | while read file; do
    ((total++))
done
echo $total                    # 0 - wrong!
```

Solution:

```bash
# Use process substitution
total=0
while read file; do
    ((total++))
done < <(ls)
echo $total                    # Correct!
```

12.6 Exercise: System Reporter

```bash
#!/bin/bash
# sysreport.sh - Generate system report using command substitution

echo "=== System Report ==="
echo "Generated: $(date)"
echo ""

# System info
echo "--- System Information ---"
echo "Hostname: $(hostname)"
echo "OS: $(uname -s)"
echo "Kernel: $(uname -r)"
echo "Uptime: $(uptime -p 2>/dev/null || uptime)"
echo ""

# CPU info
echo "--- CPU Information ---"
echo "CPUs: $(nproc)"
echo "Load: $(uptime | awk -F'load average:' '{print $2}')"
echo ""

# Memory info
echo "--- Memory Information ---"
mem_total=$(free -h | awk '/Mem:/ {print $2}')
mem_used=$(free -h | awk '/Mem:/ {print $3}')
mem_free=$(free -h | awk '/Mem:/ {print $4}')
echo "Total: $mem_total"
echo "Used: $mem_used"
```

```
echo "Free: $mem_free"
echo ""

# Disk info
echo "--- Disk Information ---"
df -h | awk 'NR==1 || /\/$/ {print}'
echo ""

# Process info
echo "--- Process Information ---"
echo "Total processes: $(ps aux | wc -l)"
echo "Running: $(ps aux | awk '$8 == "R" {print}' | wc -l)"
echo "Sleeping: $(ps aux | awk '$8 == "S" {print}' | wc -l)"
echo ""

# Current user
echo "--- User Information ---"
echo "Current user: $(whoami)"
echo "Home directory: $HOME"
echo "Shell: $SHELL"
echo ""

# File counts
echo "--- Current Directory ---"
echo "Location: $(pwd)"
echo "Files: $(find . -type f 2>/dev/null | wc -l)"
echo "Directories: $(find . -type d 2>/dev/null | wc -l)"
```

---

13 Chapter 11: Input/Output Redirection

13.1 Why This Matters in DevOps

I/O redirection manages data flow in automation:

Log Management: Separate normal output from errors, append to log files, or discard unnecessary output while keeping errors visible.

Silent Operations: Run commands silently in cron jobs while capturing errors, or suppress verbose output in CI/CD pipelines.

Multi-Level Logging: Route different log levels to different files - info to one file, errors to another, debug to a third.

Data Pipeline: Chain tools together, redirect processed data to files, or feed command output to multiple destinations.

Error Handling: Capture error messages separately, log them, and decide whether to alert or retry based on error content.

Example: `./deploy.sh 2> errors.log 1> deployment.log` - separate error logs from deployment logs for better troubleshooting.

I/O redirection controls where commands read input from and send output to. Every command has three standard streams: stdin (0), stdout (1), and stderr (2).

13.2 Understanding Streams

Every process starts with: - **stdin (0)**: Standard input (keyboard) - **stdout (1)**: Standard output (screen) - **stderr (2)**: Standard error (screen)

Redirection changes where these streams go.

13.3 Output Redirection

```
# Redirect stdout to file
echo "hello" > file.txt        # Overwrite
echo "world" >> file.txt       # Append

# Redirect stderr
command 2> errors.txt

# Redirect both stdout and stderr
command &> output.txt          # Both to same file
command > out.txt 2> err.txt   # To separate files
command 2>&1                   # stderr to stdout
```

13.4 Input Redirection

```
# Read from file
wc -l < file.txt
```

```
# Here document (inline input)
cat <<EOF
Line 1
Line 2
EOF

# Here string (one line)
grep "pattern" <<< "text to search"
```

13.5 Pipelines

Connect stdout of one command to stdin of another:

```
ls -l | grep ".txt"
cat file.txt | sort | uniq
ps aux | grep nginx | wc -l
```

13.6 Discarding Output

```
command > /dev/null            # Discard stdout
command 2> /dev/null           # Discard stderr
command &> /dev/null           # Discard both
```

`/dev/null` is a black hole that discards everything.

13.7 Custom File Descriptors

Open additional streams (3-9):

```
# Open for writing
exec 3> output.txt
echo "hello" >&3
exec 3>&-                      # Close FD 3

# Open for reading
exec 4< input.txt
read -u 4 line
exec 4<&-                      # Close FD 4
```

13.8 Exercise: Log Manager

```
#!/bin/bash
# logmanager.sh - Manage multiple log levels with file descriptors

# Setup log files
INFO_LOG="info.log"
ERROR_LOG="error.log"
DEBUG_LOG="debug.log"

# Open file descriptors
exec 3> "$INFO_LOG"            # FD 3 = info
exec 4> "$ERROR_LOG"           # FD 4 = errors
exec 5> "$DEBUG_LOG"           # FD 5 = debug

# Logging functions
log_info() {
    local msg="[$(date '+%H:%M:%S')] [INFO] $@"
    echo "$msg" >&3              # To info log
    echo "$msg"                  # To screen
}

log_error() {
    local msg="[$(date '+%H:%M:%S')] [ERROR] $@"
    echo "$msg" >&4              # To error log
    echo "$msg" >&2              # To stderr
}

log_debug() {
    local msg="[$(date '+%H:%M:%S')] [DEBUG] $@"
    echo "$msg" >&5              # To debug log
    [ "$DEBUG" = "true" ] && echo "$msg"
}

# Demo
echo "=== Log Manager Demo ==="
echo ""
```

```
log_info "Application starting"
log_debug "Initializing components"
log_info "Loading configuration"
log_debug "Config file: app.conf"
log_error "Database connection failed"
log_info "Retrying connection"
log_debug "Using backup server"
log_info "Connected successfully"

# Close file descriptors
exec 3>&-
exec 4>&-
exec 5>&-

# Show results
echo ""
echo "=== Log Files Created ==="
echo "Info log:"
cat "$INFO_LOG"
echo ""
echo "Error log:"
cat "$ERROR_LOG"
echo ""
echo "Debug log (first 3 lines):"
head -3 "$DEBUG_LOG"

# Cleanup
rm -f "$INFO_LOG" "$ERROR_LOG" "$DEBUG_LOG"
```

---

14 Chapter 12: Text Processing Tools

14.1 Why This Matters in DevOps

Text processing is fundamental for log analysis and data extraction:

Log Analysis: Search for errors, filter by severity, extract specific events, or identify patterns in

application logs.

Configuration Management: Parse config files, extract values, modify settings, or validate configuration formats.

Reporting: Generate summaries from command outputs, count occurrences, calculate statistics, or format data for dashboards.

Security Auditing: Search for security events, analyze access logs, detect suspicious patterns, or extract audit trails.

Data Transformation: Convert between formats, clean data, extract columns from structured text, or prepare data for other tools.

Example: `grep ERROR app.log | awk '{print $1, $5}' | sort | uniq -c` - count unique errors by type from application logs.

Text processing tools transform and analyze text data. The essential tools are: grep (search), sed (edit), awk (process), cut (extract), and sort/uniq (organize).

14.2 grep - Search Text

Search for patterns in text:

```
grep "error" logfile.txt       # Lines containing "error"
grep -i "error" file.txt       # Case-insensitive
grep -v "debug" file.txt       # Lines NOT containing "debug"
grep -n "error" file.txt       # Show line numbers
grep -r "TODO" .               # Recursive search
```

14.3 sed - Stream Editor

Transform text:

```
sed 's/old/new/' file.txt      # Replace first occurrence
sed 's/old/new/g' file.txt     # Replace all occurrences
sed '/pattern/d' file.txt      # Delete matching lines
sed -n '1,10p' file.txt        # Print lines 1-10
```

14.4 awk - Process Columns

Process structured data:

```
awk '{print $1}' file.txt      # Print first column
awk '{print $1, $3}' file.txt  # Print columns 1 and 3
awk '/error/ {print}' file.txt # Print matching lines
awk '{sum += $1} END {print sum}' # Sum column 1
```

`awk` splits lines into fields by whitespace. `$1` is first field, `$2` is second, etc.

14.5 cut - Extract Fields

Extract specific columns:

```
cut -d: -f1 /etc/passwd        # First field (delimiter :)
cut -d, -f1,3 data.csv         # Fields 1 and 3 from CSV
cut -c1-10 file.txt            # Characters 1-10
```

14.6 sort and uniq

Organize data:

```
sort file.txt                  # Alphabetical sort
sort -n numbers.txt            # Numeric sort
sort -r file.txt               # Reverse sort
sort -u file.txt               # Remove duplicates

sort file.txt | uniq           # Remove adjacent duplicates
sort file.txt | uniq -c        # Count occurrences
```

`uniq` only works on sorted input.

14.7 Exercise: Log Analyzer

```
#!/bin/bash
# loganalyzer.sh - Analyze log file with text processing tools

# Create sample log
cat > sample.log <<'EOF'
2024-01-01 10:00:00 INFO Server started
2024-01-01 10:01:15 ERROR Connection failed
2024-01-01 10:01:20 WARN Retrying connection
2024-01-01 10:01:25 INFO Connected successfully
2024-01-01 10:02:00 ERROR Database timeout
```

```
2024-01-01 10:02:30 ERROR Database timeout
2024-01-01 10:03:00 INFO Request processed
2024-01-01 10:03:15 WARN High memory usage
2024-01-01 10:04:00 ERROR Connection failed
2024-01-01 10:05:00 INFO Server stopped
EOF

echo "=== Log Analyzer ==="
echo ""

# Total lines
total=$(wc -l < sample.log)
echo "Total log entries: $total"
echo ""

# Count by level
echo "=== By Level ==="
echo "INFO: $(grep -c INFO sample.log)"
echo "WARN: $(grep -c WARN sample.log)"
echo "ERROR: $(grep -c ERROR sample.log)"
echo ""

# Most common errors
echo "=== Top Errors ==="
grep ERROR sample.log | \
    awk '{$1=$2=$3=""; print $0}' | \
    sort | uniq -c | sort -rn
echo ""

# Time range
echo "=== Time Range ==="
first=$(head -1 sample.log | cut -d' ' -f1-2)
last=$(tail -1 sample.log | cut -d' ' -f1-2)
echo "First: $first"
echo "Last: $last"
echo ""
```

```
# Extract only errors to new file
grep ERROR sample.log > errors_only.log
echo "Extracted $(wc -l < errors_only.log) errors to errors_only.log"
echo ""

# Show error messages
echo "=== Error Details ==="
awk '/ERROR/ {$1=$2=$3=""; print "  -"$0}' sample.log

# Cleanup
rm -f sample.log errors_only.log
```

---

15 Chapter 13: Regular Expressions

15.1 Why This Matters in DevOps

Regular expressions enable precise pattern matching and validation:

Input Validation: Validate email addresses, IP addresses, version numbers, or configuration values before processing.

Log Parsing: Extract timestamps, parse structured log entries, identify specific error patterns, or filter log events.

Configuration Validation: Ensure config files match expected formats, validate environment variable values, or check syntax.

Data Extraction: Pull specific data from command outputs, extract metrics from logs, or parse API responses.

Security: Detect malicious patterns, validate user inputs, or identify security events in audit logs.

Example: `[[ $version =~ ^v[0-9]+\.[0-9]+\.[0-9]+$ ]]` - validate semantic version format before deployment.

Regular expressions (regex) are patterns for matching text. They're essential for searching, validating, and extracting data from text.

15.2 Basic Metacharacters

- `.` - Any single character

- `^` - Start of line
- `$` - End of line
- `*` - Zero or more of previous
- `+` - One or more of previous
- `?` - Zero or one of previous
- `[]` - Character class
- `|` - OR operator

15.3 Character Classes

```
[abc]       # Matches a, b, or c
[a-z]       # Any lowercase letter
[A-Z]       # Any uppercase letter
[0-9]       # Any digit
[^0-9]      # Any non-digit
```

15.4 Quantifiers

```
a*          # Zero or more a's
a+          # One or more a's
a?          # Zero or one a
a{3}        # Exactly 3 a's
a{2,5}      # 2 to 5 a's
```

15.5 Using with grep

```
grep '^error' file.txt         # Lines starting with "error"
grep 'error$' file.txt         # Lines ending with "error"
grep '[0-9]' file.txt          # Lines containing digits
grep -E 'foo|bar' file.txt     # Lines with foo OR bar
```

Use `-E` for extended regex (no escaping needed).

15.6 Using in Bash

```
text="user@example.com"
if [[ $text =~ ^[a-z]+@[a-z]+\.[a-z]+$ ]]; then
    echo "Valid email"
fi
```

15.7 Capture Groups

```
version="v2.5.3"
if [[ $version =~ v([0-9]+)\.([0-9]+)\.([0-9]+) ]]; then
    major=${BASH_REMATCH[1]}
    minor=${BASH_REMATCH[2]}
    patch=${BASH_REMATCH[3]}
    echo "Version: $major.$minor.$patch"
fi
```

`BASH_REMATCH[0]` is the full match, `[1]` is first capture group, etc.

15.8 Common Patterns

```
# Email
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}

# IP address
([0-9]{1,3}\.){3}[0-9]{1,3}

# Date (YYYY-MM-DD)
[0-9]{4}-[0-9]{2}-[0-9]{2}

# Phone (XXX-XXX-XXXX)
[0-9]{3}-[0-9]{3}-[0-9]{4}
```

15.9 Exercise: Data Validator

```
#!/bin/bash
# validator.sh - Validate different data formats using regex

# Validation functions
validate_email() {
    local email=$1
    if [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
        return 0
    fi
    return 1
}
```

```
validate_phone() {
    local phone=$1
    if [[ $phone =~ ^[0-9]{3}-[0-9]{3}-[0-9]{4}$ ]]; then
        return 0
    fi
    return 1
}

validate_date() {
    local date=$1
    if [[ $date =~ ^[0-9]{4}-[0-9]{2}-[0-9]{2}$ ]]; then
        return 0
    fi
    return 1
}

validate_ip() {
    local ip=$1
    if [[ $ip =~ ^([0-9]{1,3}\.){3}[0-9]{1,3}$ ]]; then
        return 0
    fi
    return 1
}

# Test function
test_validation() {
    local type=$1
    local value=$2

    printf "%-15s: %-25s " "$type" "$value"

    case $type in
        Email)
            validate_email "$value" && echo "  Valid" || echo "  Invalid"
            ;;
        Phone)
```

```
            validate_phone "$value" && echo "  Valid" || echo "  Invalid"
            ;;
        Date)
            validate_date "$value" && echo "  Valid" || echo "  Invalid"
            ;;
        IP)
            validate_ip "$value" && echo "  Valid" || echo "  Invalid"
            ;;
    esac
}

# Run tests
echo "=== Data Validator ==="
echo ""

test_validation "Email" "user@example.com"
test_validation "Email" "invalid.email"
test_validation "Email" "user@domain.co.uk"

echo ""
test_validation "Phone" "555-123-4567"
test_validation "Phone" "5551234567"
test_validation "Phone" "555-12-34567"

echo ""
test_validation "Date" "2024-01-15"
test_validation "Date" "2024/01/15"
test_validation "Date" "24-01-15"

echo ""
test_validation "IP" "192.168.1.1"
test_validation "IP" "256.1.1.1"
test_validation "IP" "10.0.0.1"
```

16 Chapter 14: Exit Status and Error Handling

16.1 Why This Matters in DevOps

Proper error handling prevents failures from cascading:

Deployment Safety: Stop deployment immediately if any step fails, preventing deployment of broken code or misconfigured services.

CI/CD Reliability: Fail builds when tests don't pass, block merges on linting errors, or abort pipelines on security scan failures.

Defensive Scripting: Check prerequisites before operations, validate inputs before processing, or verify state before making changes.

Meaningful Feedback: Provide clear error messages, exit with specific codes for different failures, or log context for troubleshooting.

Automation Resilience: Handle expected failures gracefully, retry transient errors, or clean up resources even when operations fail.

Example: `set -e; validate_config || exit 1; backup_current || exit 2; deploy_new` - each failure type has distinct exit code for debugging.

Exit status indicates whether a command succeeded (0) or failed (non-zero). Proper error handling makes scripts reliable and prevents cascading failures.

16.2 Exit Status Basics

Every command returns an exit code: - `0` = success - `1-255` = failure

Check with `$?`:

```
ls /tmp
echo $?           # 0 (success)

ls /nonexistent
echo $?           # 2 (failure)
```

16.3 Setting Exit Status

```
#!/bin/bash
if [ ! -f "$1" ]; then
    echo "Error: File not found"
    exit 1
```

```
fi

echo "Processing $1"
exit 0
```

Always exit with appropriate codes.

16.4 Conditional Execution

Chain commands based on success:

```
# AND - run if previous succeeds
mkdir temp && cd temp && touch file.txt

# OR - run if previous fails
cd /nonexistent || echo "Failed to change directory"

# Combine
cd project && make || echo "Build failed"
```

16.5 Error Handling with set

```
set -e              # Exit immediately on error
set -u              # Exit on undefined variable
set -o pipefail     # Exit if any pipe command fails

# Combined
set -euo pipefail
```

These options make scripts fail-fast instead of continuing with errors.

16.6 Cleanup on Exit

Always cleanup temporary files:

```
#!/bin/bash
temp=$(mktemp)

cleanup() {
    rm -f "$temp"
}
```

```
trap cleanup EXIT

# Script continues...
# cleanup runs automatically on exit
```

16.7 Exercise: Robust Backup Script

```
#!/bin/bash
# backup.sh - Robust backup with proper error handling

set -euo pipefail

# Configuration
SOURCE_DIR="${1:-.}"
BACKUP_DIR="$HOME/backups"
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
BACKUP_FILE="$BACKUP_DIR/backup_$TIMESTAMP.tar.gz"

# Error codes
readonly E_SUCCESS=0
readonly E_NO_SOURCE=1
readonly E_BACKUP_FAILED=2
readonly E_VERIFY_FAILED=3

# Error handler
error() {
    echo "ERROR: $1" >&2
    exit "${2:-1}"
}

# Cleanup
cleanup() {
    if [ -f "$BACKUP_FILE.tmp" ]; then
        rm -f "$BACKUP_FILE.tmp"
    fi
}
```

```
trap cleanup EXIT

# Validate source
echo "=== Backup Script ==="
echo ""

if [ ! -d "$SOURCE_DIR" ]; then
    error "Source directory not found: $SOURCE_DIR" $E_NO_SOURCE
fi

echo "Source: $SOURCE_DIR"
echo "Backup: $BACKUP_FILE"
echo ""

# Create backup directory
mkdir -p "$BACKUP_DIR" || error "Cannot create backup directory" $E_BACKUP_FAILED

# Create backup
echo "Creating backup..."
if ! tar -czf "$BACKUP_FILE.tmp" -C "$(dirname "$SOURCE_DIR")" "$(basename "$SOURCE_DIR")" 2>/
    error "Backup creation failed" $E_BACKUP_FAILED
fi

# Move to final location (atomic)
mv "$BACKUP_FILE.tmp" "$BACKUP_FILE"

# Verify backup
echo "Verifying backup..."
if ! tar -tzf "$BACKUP_FILE" > /dev/null 2>&1; then
    error "Backup verification failed" $E_VERIFY_FAILED
fi

# Report
size=$(du -h "$BACKUP_FILE" | cut -f1)
files=$(tar -tzf "$BACKUP_FILE" | wc -l)

echo ""
```

```
echo "=== Backup Complete ==="
echo "File: $BACKUP_FILE"
echo "Size: $size"
echo "Files: $files"

exit $E_SUCCESS
```

Test it:

```
chmod +x backup.sh
./backup.sh /tmp              # Backup /tmp
./backup.sh /nonexistent      # Should show error
```

---

17 Chapter 15: Signals and Traps

17.1 Why This Matters in DevOps

Signal handling ensures graceful shutdown and proper cleanup:

Resource Cleanup: Always clean up temp files, close database connections, or release locks even when scripts are interrupted.

Graceful Shutdown: Allow running operations to complete before exiting, save state, or notify monitoring systems.

Long-Running Services: Handle reload signals to refresh configuration without downtime, or respond to shutdown requests cleanly.

Deployment Safety: Trap errors during deployment to rollback changes, or clean up partially deployed resources.

Container Lifecycle: Handle SIGTERM from Kubernetes to shut down gracefully within termination grace period.

Example: `trap cleanup EXIT; trap reload HUP` - always cleanup on exit, reload config on HUP signal without restart.

Signals are notifications sent to processes. Traps catch signals to perform cleanup, ignore interrupts, or handle events gracefully.

17.2 Common Signals

- `SIGINT (2)` - Ctrl+C
- `SIGTERM (15)` - Termination request
- `SIGKILL (9)` - Force kill (cannot be trapped)
- `SIGHUP (1)` - Hangup/reload config
- `EXIT (0)` - Script exit (pseudo-signal)

17.3 Basic Trap

Catch signals and run commands:

```
#!/bin/bash

cleanup() {
    echo "Cleaning up..."
    rm -f /tmp/myfile
}

trap cleanup EXIT      # Run cleanup when script exits

# Script work...
```

17.4 Ignoring Signals

Prevent interruption during critical operations:

```
trap '' INT TERM       # Ignore Ctrl+C and TERM

# Critical work that cannot be interrupted
echo "Processing..."
sleep 5

trap - INT TERM        # Restore default behavior
```

17.5 Graceful Shutdown

```
#!/bin/bash

running=true
```

```
shutdown() {
    echo "Shutdown requested..."
    running=false
}

trap shutdown INT TERM

while $running; do
    echo "Working..."
    sleep 1
done

echo "Exiting gracefully"
```

17.6 Configuration Reload

Use `SIGHUP` to reload without restart:

```
#!/bin/bash

CONFIG_FILE="app.conf"

load_config() {
    source "$CONFIG_FILE"
    echo "Config loaded"
}

trap load_config HUP
load_config

while true; do
    echo "Running with port $PORT"
    sleep 5
done

# From another terminal: kill -HUP $PID
```

17.7 Exercise: Service Manager

```
#!/bin/bash
# service.sh - Simple service with signal handling

PID_FILE="/tmp/service.pid"
LOG_FILE="/tmp/service.log"
running=true
config_loaded=false

# Logging
log() {
    echo "[$(date '+%H:%M:%S')] $*" | tee -a "$LOG_FILE"
}

# Load configuration
load_config() {
    # Simulate config loading
    INTERVAL=2
    MAX_COUNT=100
    config_loaded=true
    log "Configuration loaded (interval=$INTERVAL)"
}

# Shutdown handler
shutdown() {
    log "Shutdown signal received"
    running=false
}

# Cleanup
cleanup() {
    log "Cleaning up..."
    rm -f "$PID_FILE"
    log "Service stopped"
}
```

```
# Setup traps
trap shutdown INT TERM
trap load_config HUP
trap cleanup EXIT

# Initialize
echo "=== Service Manager ===" | tee "$LOG_FILE"
load_config
echo $$ > "$PID_FILE"
log "Service started (PID: $$)"

# Main loop
count=0
while $running && [ $count -lt $MAX_COUNT ]; do
    ((count++))
    log "Processing task $count"
    sleep $INTERVAL
done

log "Service exiting normally"
```

Test it:

```
chmod +x service.sh

# Terminal 1: Run service
./service.sh

# Terminal 2: Send signals
PID=$(cat /tmp/service.pid)
kill -HUP $PID      # Reload config
kill -TERM $PID     # Graceful shutdown

# View log
cat /tmp/service.log
```

---

18 Chapter 16: Parallel Processing

18.1 Why This Matters in DevOps

Parallel processing dramatically reduces execution time:

Multi-Server Deployments: Deploy to 10 servers in parallel instead of sequentially - 10x faster deployments.

Batch Processing: Process hundreds of files, images, or logs concurrently using available CPU cores.

Health Checks: Check multiple endpoints or services simultaneously rather than waiting for each sequentially.

Data Collection: Gather metrics from multiple sources in parallel, or fetch logs from multiple servers concurrently.

CI/CD Speed: Run tests in parallel, build multiple components simultaneously, or deploy to multiple regions at once.

Example: `xargs -P 10 -I {} deploy.sh {}` - deploy to 10 servers simultaneously instead of one at a time.

Parallel processing runs multiple tasks simultaneously, dramatically improving performance for independent operations. Use background jobs, `wait`, and `xargs` for parallelization.

18.2 Background Jobs

Run commands in background with `&`:

```
command1 &
command2 &
wait              # Wait for all to complete
```

18.3 Limiting Concurrent Jobs

```
#!/bin/bash
MAX_JOBS=4

for file in *.txt; do
    # Wait if too many jobs
    while [ $(jobs -r | wc -l) -ge $MAX_JOBS ]; do
        sleep 0.1
```

```
    done

    # Process in background
    process_file "$file" &
done

wait              # Wait for all to complete
```

18.4 xargs for Parallel Processing

`xargs` runs commands in parallel:

```
# Process 4 files at once
find . -name "*.txt" | xargs -P 4 -I {} process {}

# Use all CPU cores
find . -name "*.log" | xargs -P $(nproc) gzip

# Parallel downloads
cat urls.txt | xargs -P 5 wget
```

`-P N` runs N processes simultaneously.

18.5 Parallel Function Execution

```
#!/bin/bash

process_file() {
    local file=$1
    echo "Processing $file"
    sleep 1
    echo "Done: $file"
}

export -f process_file

# Run in parallel
ls *.txt | xargs -P 4 -I {} bash -c 'process_file "{}"'
```

18.6 Exercise: Parallel File Processor

```
#!/bin/bash
# parallel_process.sh - Process files in parallel with progress

MAX_JOBS=4
PROCESSED=0
TOTAL=0

# Setup test files
echo "Creating test files..."
mkdir -p test_files
for i in {1..20}; do
    echo "Sample content $i" > "test_files/file$i.txt"
done

TOTAL=$(find test_files -name "*.txt" | wc -l)
echo "Total files to process: $TOTAL"
echo ""

# Process function
process_file() {
    local file=$1
    local duration=$((RANDOM % 3 + 1))

    echo "  Processing: $file (${duration}s)"
    sleep $duration

    # Simulate processing
    wc -l "$file" > /dev/null

    echo "    Complete: $file"
}

export -f process_file

# Start time
```

```
START=$(date +%s)

# Process in parallel
echo "=== Processing Files ==="
find test_files -name "*.txt" | \
    xargs -P $MAX_JOBS -I {} bash -c 'process_file "{}"'

# End time
END=$(date +%s)
DURATION=$((END - START))

# Report
echo ""
echo "=== Processing Complete ==="
echo "Files processed: $TOTAL"
echo "Time taken: ${DURATION}s"
echo "Average: $((DURATION / TOTAL))s per file"

# Calculate theoretical sequential time
THEORETICAL=$((TOTAL * 2))
SPEEDUP=$(echo "scale=2; $THEORETICAL / $DURATION" | bc)
echo "Speedup: ${SPEEDUP}x faster"

# Cleanup
echo ""
read -p "Remove test files? (y/n) " answer
[ "$answer" = "y" ] && rm -rf test_files
```

Test it:

```
chmod +x parallel_process.sh
./parallel_process.sh
```

---

19 Chapter 17: Debugging

19.1 Why This Matters in DevOps

Effective debugging reduces downtime and speeds problem resolution:

Production Troubleshooting: Quickly identify why deployments fail, diagnose automation issues, or understand unexpected behavior.

Development Speed: Find bugs faster during script development, validate logic, or understand complex interactions.

CI/CD Debugging: Trace why pipeline steps fail, understand variable values at each stage, or verify execution flow.

Operational Insights: Add debug flags to production scripts for detailed logging when issues occur, without always showing verbose output.

Team Collaboration: Debug output helps teammates understand what scripts do, making maintenance and handoffs easier.

Example: `DEBUG=true ./deploy.sh` - enable detailed logging only when debugging, keeping normal runs clean and fast.

Debugging helps identify and fix script errors. Use built-in debug modes, strategic logging, and systematic approaches to troubleshoot issues.

19.2 Debug Mode

Enable trace mode to see each command:

```
bash -x script.sh            # Debug entire script

# Or in script:
#!/bin/bash -x

# Or selectively:
set -x                       # Enable
commands
set +x                       # Disable
```

Trace mode shows each command before executing it.

19.3 Syntax Check

Check syntax without running:

```
bash -n script.sh            # Check syntax only
```

19.4 Custom Debug Output

```
DEBUG=${DEBUG:-false}

debug() {
    if [ "$DEBUG" = "true" ]; then
        echo "[DEBUG] $*" >&2
    fi
}

debug "Starting processing"
debug "Variable x = $x"
```

Run with: `DEBUG=true ./script.sh`

19.5 PS4 Prompt

Customize debug output:

```
export PS4='+ ${LINENO}: '
set -x

# Shows line numbers:
# + 10: command
```

19.6 Common Debugging Techniques

1. Echo debugging:

```
echo "About to process $file"
process_file "$file"
echo "Finished processing $file"
```

2. Verify assumptions:

```
echo "File exists: $([ -f "$file" ] && echo yes || echo no)"
echo "Variable empty: $([ -z "$var" ] && echo yes || echo no)"
```

3. Check exit codes:

```
command
echo "Exit code: $?"
```

19.7 Error Handler

Trap errors and show context:

```
error_handler() {
    echo "Error on line $1" >&2
    exit 1
}

trap 'error_handler $LINENO' ERR
```

19.8 Exercise: Debug Helper

```
#!/bin/bash
# debugger.sh - Script with debugging features

# Debug configuration
DEBUG=${DEBUG:-false}
TRACE=${TRACE:-false}

# Enable features
[ "$TRACE" = "true" ] && set -x

# Debug function
debug() {
    [ "$DEBUG" = "true" ] && echo "[DEBUG] $*" >&2
}

# Timer
start_timer() {
    TIMER_START=$(date +%s%N)
}

end_timer() {
```

```
    local end=$(date +%s%N)
    local elapsed=$(( (end - TIMER_START) / 1000000 ))
    debug "Operation took ${elapsed}ms"
}

# Validate function
validate() {
    local var_name=$1
    local var_value=${!var_name}

    if [ -z "$var_value" ]; then
        echo "ERROR: $var_name is not set" >&2
        return 1
    fi

    debug "$var_name = '$var_value'"
    return 0
}

# Main program
main() {
    echo "=== Debug Helper Demo ==="
    echo ""

    # Set some variables
    export NAME="TestApp"
    export VERSION="1.0"

    # Validate
    debug "Validating configuration..."
    validate NAME || exit 1
    validate VERSION || exit 1
    debug "Validation complete"

    # Timed operation
    echo "Processing data..."
    start_timer
```

```
    for i in {1..5}; do
        debug "Processing item $i"
        sleep 0.1
    done

    end_timer
    echo "Processing complete"

    # Show environment
    if [ "$DEBUG" = "true" ]; then
        echo ""
        echo "=== Environment ==="
        echo "NAME: $NAME"
        echo "VERSION: $VERSION"
        echo "PWD: $PWD"
    fi
}

main
```

Test it:

```
chmod +x debugger.sh

# Normal mode
./debugger.sh

# Debug mode
DEBUG=true ./debugger.sh

# Trace mode
TRACE=true ./debugger.sh

# Both
DEBUG=true TRACE=true ./debugger.sh
```

---

20 Chapter 18: Best Practices

20.1 Why This Matters in DevOps

Professional scripting standards are crucial for team success:

Production Reliability: Well-written scripts fail safely, handle errors properly, and don't cause outages or data loss.

Team Collaboration: Consistent style and structure make scripts readable and maintainable by the whole team, not just the original author.

Security: Proper input validation, secure defaults, and safe operations prevent security vulnerabilities in automation.

Operational Excellence: Scripts with good logging, error messages, and documentation are easier to troubleshoot when things go wrong.

Long-term Maintenance: Following best practices means scripts remain useful and maintainable months or years later.

Example: Scripts following best practices with `set -euo pipefail`, input validation, and proper error handling catch problems early rather than causing mysterious failures.

Best practices ensure scripts are reliable, maintainable, and professional. Follow these guidelines for production-quality code.

20.2 Script Structure

Every script should have:

```
#!/bin/bash
# Script: scriptname.sh
# Purpose: What it does
# Author: Your name
# Date: 2024-01-01

set -euo pipefail           # Strict mode
IFS=$'\n\t'                 # Safe word splitting

# Constants
readonly SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
readonly VERSION="1.0.0"
```

```
# Configuration with defaults
CONFIG_FILE="${CONFIG_FILE:-/etc/app.conf}"
DEBUG="${DEBUG:-false}"

# Functions
main() {
    # Main logic
}

# Run
main "$@"
```

20.3 Variable Practices

Always quote:

```
echo "$variable"            # Good
echo $variable              # Bad - word splitting
```

Use braces:

```
echo "${var}_suffix"        # Good
echo "$var_suffix"          # Wrong variable
```

Readonly constants:

```
readonly MAX_RETRIES=3
readonly CONFIG_FILE="/etc/app.conf"
```

Local in functions:

```
my_function() {
    local temp="value"      # Local scope
    echo "$temp"
}
```

20.4 Error Handling

Validate input:

```
[ $# -eq 0 ] && { echo "Usage: $0 <file>"; exit 1; }
[ ! -f "$1" ] && { echo "File not found"; exit 1; }
```

Check commands exist:

```
command -v git >/dev/null || { echo "git not installed"; exit 1; }
```

Use exit codes:

```
readonly E_SUCCESS=0
readonly E_BADARGS=1
readonly E_NOFILE=2

[ ! -f "$file" ] && exit $E_NOFILE
```

20.5 Security

Validate paths:

```
# Prevent directory traversal
realpath="$(realpath "$user_path")"
[[ "$realpath" =~ ^/safe/dir ]] || exit 1
```

Use – for arguments:

```
rm -- "$file"               # Handles files starting with -
```

Never eval user input:

```
eval "$user_input"          # NEVER!

# Instead:
case $user_input in
    start|stop) $user_input;;
    *) echo "Invalid";;
esac
```

20.6 Performance

Prefer built-ins:

```
[[ "$str" == *pattern* ]]   # Good - built-in
echo "$str" | grep pattern  # Slow - subprocess
```

Read files efficiently:

```
while IFS= read -r line; do # Good
    process "$line"
```

```
done < file

for line in $(cat file); do # Bad
    process "$line"
done
```

20.7 Exercise: Production Script Template

```
#!/bin/bash
# production_template.sh - Production-ready script template

set -euo pipefail
IFS=$'\n\t'

# Metadata
readonly SCRIPT_NAME="$(basename "$0")"
readonly SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
readonly VERSION="1.0.0"

# Exit codes
readonly E_SUCCESS=0
readonly E_BADARGS=1
readonly E_NOFILE=2

# Configuration
LOG_FILE="${LOG_FILE:-/tmp/${SCRIPT_NAME}.log}"
DEBUG="${DEBUG:-false}"

# Logging
log() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $*" | tee -a "$LOG_FILE"
}

error() {
    log "ERROR: $*" >&2
    exit "${2:-1}"
}
```

```
debug() {
    [ "$DEBUG" = "true" ] && log "DEBUG: $*"
}

# Cleanup
cleanup() {
    debug "Cleanup called"
    [ -f "$temp_file" ] && rm -f "$temp_file"
}

trap cleanup EXIT

# Help
show_help() {
    cat << EOF
$SCRIPT_NAME v$VERSION

Usage: $SCRIPT_NAME [OPTIONS] FILE

Process a file with validation and error handling.

OPTIONS:
    -h, --help      Show this help
    -v, --version   Show version
    -d, --debug     Enable debug output

EXAMPLES:
    $SCRIPT_NAME input.txt
    DEBUG=true $SCRIPT_NAME input.txt

EOF
    exit 0
}

# Parse arguments
parse_args() {
```

```
    while [ $# -gt 0 ]; do
        case $1 in
            -h|--help) show_help;;
            -v|--version) echo "$VERSION"; exit 0;;
            -d|--debug) DEBUG=true;;
            --) shift; break;;
            -*) error "Unknown option: $1" $E_BADARGS;;
            *) break;;
        esac
        shift
    done

    # Validate required arguments
    [ $# -eq 0 ] && error "No file specified" $E_BADARGS
    [ ! -f "$1" ] && error "File not found: $1" $E_NOFILE

    FILE="$1"
}

# Main processing
process_file() {
    local file=$1

    log "Processing $file"
    debug "File size: $(wc -c < "$file") bytes"

    # Create temp file
    temp_file=$(mktemp) || error "Cannot create temp file"
    debug "Temp file: $temp_file"

    # Process (example: remove comments and empty lines)
    grep -v '^#' "$file" | grep -v '^$' > "$temp_file"

    # Show results
    local original=$(wc -l < "$file")
    local processed=$(wc -l < "$temp_file")
    log "Processed $original lines -> $processed lines"
```

```
    # Display
    echo ""
    echo "=== Processed Content ==="
    cat "$temp_file"
}

# Main
main() {
    log "Starting $SCRIPT_NAME v$VERSION"

    parse_args "$@"
    process_file "$FILE"

    log "Complete"
    exit $E_SUCCESS
}

# Run
main "$@"
```

Test it:

```
# Create test file
cat > test.txt <<EOF
# This is a comment
Line 1
Line 2
# Another comment

Line 3
EOF

chmod +x production_template.sh

# Run
./production_template.sh test.txt
./production_template.sh -d test.txt
```

```
./production_template.sh --help
```

---

21 Conclusion

You've learned the essential concepts of shell scripting:

21.1 Core Skills

- Environment variables for configuration
- Command arguments for flexibility
- Wildcards for file operations
- String manipulation for text processing
- Arithmetic for calculations
- Conditionals for decision making
- Loops for repetition
- Functions for modularity
- Arrays for structured data

21.2 Advanced Techniques

- Command substitution for capturing output
- I/O redirection for stream control
- Text processing with grep, sed, awk
- Regular expressions for pattern matching
- Error handling for reliability
- Signal handling for graceful shutdown
- Parallel processing for performance
- Debugging for troubleshooting
- Best practices for professional code

21.3 What's Next

Start small and build up: 1. Automate repetitive tasks 2. Write utility scripts for daily work 3. Create deployment automation 4. Build monitoring tools 5. Contribute to open source

21.4 Key Principles

- **Simplicity**: Keep scripts simple and focused

- **Validation**: Always check inputs and conditions
- **Error handling**: Fail gracefully with clear messages
- **Documentation**: Comment complex logic
- **Testing**: Test with various inputs
- **Maintenance**: Write code you'll understand later

21.5 Resources

- `man bash` - Bash manual
- shellcheck.net - Script analysis
- github.com/awesome-shell - Curated resources

Shell scripting is a fundamental skill that improves with practice. Write scripts, read others' code, and continuously refine your approach.

Happy scripting!