Complete Bash Scripting Guide

Table of Contents

Introduction and Setup

What is Bash?

Bash (Bourne Again SHell) is a command-line interpreter and scripting language. It's the default shell on most Linux distributions and macOS.

Checking Your Bash Version

```bash
```

```bash
# Check Bash version
bash --version
echo $BASH_VERSION

# Check which Bash you're using
which bash
echo $0
```

Script Permissions

```bash
# Make script executable
chmod +x script.sh

# Different permission levels
chmod 755 script.sh  # Owner: read/write/execute, Others: read/execute
chmod 744 script.sh  # Owner: read/write/execute, Others: read only
chmod u+x script.sh  # Add execute permission for owner
```

Basic Script Structure

The Shebang Line

```bash
#!/bin/bash
# The shebang tells the system which interpreter to use

#!/usr/bin/env bash
# More portable - finds bash in PATH

#!/bin/bash -e
# Exit immediately if a command exits with non-zero status

#!/bin/bash -x
# Print commands and their arguments as they are executed
```

Basic Script Template

```bash
```

```bash
#!/bin/bash

# Script: example.sh
# Purpose: Demonstrate basic Bash script structure
# Author: Your Name
# Date: $(date)
# Version: 1.0

# Exit on any error
set -e

# Exit on undefined variables
set -u

# Exit on pipe failures
set -o pipefail

# Enable debug mode (optional)
# set -x

# Main script logic goes here
echo "Hello, World!"

# Exit with success code
exit 0
```

Comments

```bash
# Single line comment

: '
Multi-line comment
Everything between the quotes
is treated as a comment
'

# Inline comment
echo "Hello" # This prints Hello
```

Variables and Data Types

Variable Declaration and Assignment

bash

```bash
# Variable assignment (no spaces around =)
name="John Doe"
age=30
PI=3.14159

# Using variables
echo "Name: $name"
echo "Age: ${age}"
echo "Value of PI: $PI"

# Command substitution
current_date=$(date)
files_count=`ls | wc -l`  # Old syntax, prefer $()

echo "Today is: $current_date"
echo "Number of files: $files_count"
```

Variable Types and Scope

bash

```bash
# Local variables (default)
local_var="I'm local"

# Global variables
declare -g global_var="I'm global"

# Read-only variables
declare -r readonly_var="Cannot change me"
readonly another_readonly="Also cannot change"

# Integer variables
declare -i number=42
number="50"  # This works
# number="hello"  # This would set number to 0

# Array variables
declare -a my_array=("apple" "banana" "cherry")

# Associative arrays (Bash 4.0+)
declare -A assoc_array=([key1]="value1" [key2]="value2")

# Export variables (available to child processes)
export PATH="/usr/local/bin:$PATH"
export DATABASE_URL="postgresql://localhost/mydb"
```

Special Variables

bash

```bash
#!/bin/bash

echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
echo "All arguments as single string: $*"
echo "Number of arguments: $#"
echo "Exit status of last command: $?"
echo "Process ID of script: $$"
echo "Process ID of last background command: $!"

# Example usage: ./script.sh arg1 arg2 arg3
```

Variable Expansion and Manipulation

bash

```bash
name="john_doe"

# Basic expansion
echo $name
echo ${name}

# Default values
echo ${name:-"default"}      # Use "default" if name is unset
echo ${name:="default_value"} # Set name to "default_value" if unset
echo ${name:?"Error message"} # Error if name is unset
echo ${name:+"alternative"}   # Use "alternative" if name is set

# String length
echo ${#name} # Length of string

# Substring extraction
echo ${name:0:4}   # Extract 4 characters starting from position 0
echo ${name:5}     # Extract from position 5 to end
echo ${name: -3}   # Extract last 3 characters (note the space)

# Pattern matching
filename="document.txt"
echo ${filename%.txt}    # Remove shortest match of .txt from end
echo ${filename%.*}      # Remove shortest match of .* from end
echo ${filename%%.*}     # Remove longest match of .* from end
echo ${filename#*/}      # Remove shortest match of */ from beginning
echo ${filename##*/}     # Remove longest match of */ from beginning

# Case conversion (Bash 4.0+)
text="Hello World"
echo ${text,,}   # Convert to lowercase: hello world
echo ${text^^}   # Convert to uppercase: HELLO WORLD
echo ${text^}    # Capitalize first letter: Hello world
echo ${text,}    # Lowercase first letter: hello World
```

Input and Output

Reading User Input

bash

```bash
# Basic input
echo "Enter your name:"
read name
echo "Hello, $name!"

# Input with prompt
read -p "Enter your age: " age
echo "You are $age years old"

# Silent input (for passwords)
read -s -p "Enter password: " password
echo  # New line after password
echo "Password entered"

# Reading with timeout
if read -t 10 -p "Enter something (10 seconds): " input; then
  echo "You entered: $input"
else
  echo "Timeout occurred"
fi

# Reading single character
read -n 1 -p "Press any key to continue..." key
echo

# Reading from file
while IFS= read -r line; do
  echo "Line: $line"
done < "input.txt"

# Reading multiple values
read -p "Enter name and age: " name age
echo "Name: $name, Age: $age"
```

Output and Formatting

bash

```bash
# Basic output
echo "Simple output"
printf "Formatted output: %s is %d years old\n" "John" 30

# Formatted output examples
printf "%-10s %5d %8.2f\n" "Name" 123 45.678
printf "%s\n" "Line 1" "Line 2" "Line 3"

# Output redirection
echo "This goes to stdout"
echo "This goes to stderr" >&2

# Here documents
cat << EOF
This is a multi-line
string that can contain
variables like $USER
EOF

# Here strings
cat <<< "This is a here string"
```

Colors and Formatting

```bash
```

```bash
# ANSI color codes
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
PURPLE='\033[0;35m'
CYAN='\033[0;36m'
WHITE='\033[1;37m'
NC='\033[0m' # No Color

echo -e "${RED}This is red text${NC}"
echo -e "${GREEN}This is green text${NC}"
echo -e "${YELLOW}This is yellow text${NC}"

# Text formatting
BOLD='\033[1m'
UNDERLINE='\033[4m'
BLINK='\033[5m'

echo -e "${BOLD}Bold text${NC}"
echo -e "${UNDERLINE}Underlined text${NC}"

# Function for colored output
print_color() {
  local color=$1
  local message=$2
  echo -e "${color}${message}${NC}"
}

print_color "$RED" "Error: Something went wrong"
print_color "$GREEN" "Success: Operation completed"
```

Conditional Statements

If Statements

bash

```bash
# Basic if statement
if [ "$age" -gt 18 ]; then
  echo "You are an adult"
fi

# If-else statement
if [ "$age" -gt 18 ]; then
  echo "You are an adult"
else
  echo "You are a minor"
fi

# If-elif-else statement
if [ "$age" -lt 13 ]; then
  echo "You are a child"
elif [ "$age" -lt 18 ]; then
  echo "You are a teenager"
else
  echo "You are an adult"
fi

# Multiple conditions
if [ "$age" -gt 18 ] && [ "$age" -lt 65 ]; then
  echo "You are a working-age adult"
fi

# Using [[ ]] (preferred for Bash)
if [[ "$name" == "John" && "$age" -gt 25 ]]; then
  echo "Hello John, you're over 25"
fi
```

Test Operators

bash

```
# Numeric comparisons
[ "$a" -eq "$b" ]  # Equal
[ "$a" -ne "$b" ]  # Not equal
[ "$a" -lt "$b" ]  # Less than
[ "$a" -le "$b" ]  # Less than or equal
[ "$a" -gt "$b" ]  # Greater than
[ "$a" -ge "$b" ]  # Greater than or equal

# String comparisons
[ "$str1" = "$str2" ]  # Equal (POSIX)
[ "$str1" == "$str2" ]  # Equal (Bash)
[ "$str1" != "$str2" ]  # Not equal
[ "$str1" \< "$str2" ]  # Less than (lexicographic)
[ "$str1" \> "$str2" ]  # Greater than (lexicographic)
[ -n "$str" ]          # String is not empty
[ -z "$str" ]          # String is empty

# File tests
[ -e "$file" ]    # File exists
[ -f "$file" ]    # Is a regular file
[ -d "$file" ]    # Is a directory
[ -r "$file" ]    # Is readable
[ -w "$file" ]    # Is writable
[ -x "$file" ]    # Is executable
[ -s "$file" ]    # File is not empty
[ -L "$file" ]    # Is a symbolic link

# Advanced pattern matching with [[]]
[[ "$string" =~ ^[0-9]+$ ]]    # Matches digits only
[[ "$string" == *.txt ]]       # Ends with .txt
[[ "$string" != *[[:space:]]* ]] # No whitespace
```

Case Statements

bash

```bash
# Basic case statement
read -p "Enter a letter: " letter
case "$letter" in
  [aeiou])
    echo "You entered a vowel"
    ;;
  [bcdfghjklmnpqrstvwxyz])
    echo "You entered a consonant"
    ;;
  [0-9])
    echo "You entered a number"
    ;;
  *)
    echo "You entered something else"
    ;;
esac

# Case with multiple patterns
case "$1" in
  start|begin|run)
    echo "Starting the service..."
    ;;
  stop|end|quit)
    echo "Stopping the service..."
    ;;
  restart|reload)
    echo "Restarting the service..."
    ;;
  status|info)
    echo "Checking service status..."
    ;;
  *)
    echo "Usage: $0 {start|stop|restart|status}"
    exit 1
    ;;
esac
```

Loops

For Loops

bash

```bash
# Basic for loop
for i in 1 2 3 4 5; do
  echo "Number: $i"
done

# For loop with range
for i in {1..10}; do
  echo "Count: $i"
done

# For loop with step
for i in {0..20..2}; do
  echo "Even number: $i"
done

# For loop with array
fruits=("apple" "banana" "cherry" "date")
for fruit in "${fruits[@]}"; do
  echo "Fruit: $fruit"
done

# For loop with command output
for file in $(ls *.txt); do
  echo "Processing: $file"
done

# For loop with globbing
for file in *.log; do
  if [ -f "$file" ]; then
    echo "Log file: $file"
  fi
done

# C-style for loop
for ((i=1; i<=10; i++)); do
  echo "Iteration: $i"
done

# Nested for loops
for ((i=1; i<=3; i++)); do
  for ((j=1; j<=3; j++)); do
    echo "i=$i, j=$j"
```

```
  done
done
```

While Loops

bash

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

# While loop reading file
while IFS= read -r line; do
  echo "Line: $line"
done < "input.txt"

# While loop with user input
while true; do
  read -p "Enter command (quit to exit): " cmd
  case "$cmd" in
    quit)
      break
      ;;
    *)
      echo "You entered: $cmd"
      ;;
  esac
done

# While loop with condition check
while [[ ! -f "important_file.txt" ]]; do
  echo "Waiting for file to be created..."
  sleep 1
done
echo "File found!"
```

Until Loops

bash

```bash
# Basic until loop (opposite of while)
count=1
until [ $count -gt 5 ]; do
  echo "Count: $count"
  ((count++))
done

# Until loop waiting for condition
until ping -c 1 google.com &> /dev/null; do
  echo "Waiting for internet connection..."
  sleep 5
done
echo "Internet connection established!"
```

Loop Control

bash

```bash
# Break and continue
for i in {1..10}; do
  if [ $i -eq 3 ]; then
    continue  # Skip iteration when i=3
  fi
  if [ $i -eq 8 ]; then
    break    # Exit loop when i=8
  fi
  echo "Number: $i"
done

# Breaking out of nested loops
for ((i=1; i<=3; i++)); do
  for ((j=1; j<=3; j++)); do
    if [ $i -eq 2 ] && [ $j -eq 2 ]; then
      break 2  # Break out of both loops
    fi
    echo "i=$i, j=$j"
  done
done
```

Functions

Basic Function Definition

bash

```bash
# Method 1: function keyword
function greet() {
  echo "Hello, $1!"
}

# Method 2: without function keyword (preferred)
greet() {
  echo "Hello, $1!"
}

# Calling the function
greet "World"
greet "Alice"
```

Function Parameters and Return Values

bash

```bash
# Function with multiple parameters
calculate_area() {
  local length=$1
  local width=$2
  local area=$((length * width))
  echo $area
}

# Using the function
result=$(calculate_area 10 5)
echo "Area: $result"

# Function with return value
is_even() {
  local number=$1
  if [ $((number % 2)) -eq 0 ]; then
    return 0  # Success (true)
  else
    return 1  # Failure (false)
  fi
}

# Using return value
if is_even 4; then
  echo "4 is even"
else
  echo "4 is odd"
fi
```

Local Variables and Scope

bash

```bash
global_var="I'm global"

test_scope() {
  local local_var="I'm local"
  global_var="Modified global"
  echo "Inside function:"
  echo "  Local: $local_var"
  echo "  Global: $global_var"
}

echo "Before function: $global_var"
test_scope
echo "After function: $global_var"
# echo "Local var: $local_var"  # This would cause an error
```

Advanced Function Features

bash

```bash
# Function with variable arguments
sum_numbers() {
  local total=0
  for num in "$@"; do
    total=$((total + num))
  done
  echo $total
}

result=$(sum_numbers 1 2 3 4 5)
echo "Sum: $result"

# Function that modifies global array
add_to_list() {
  local item=$1
  global_list+=("$item")
}

global_list=()
add_to_list "apple"
add_to_list "banana"
echo "List: ${global_list[@]}"

# Recursive function
factorial() {
  local n=$1
  if [ $n -le 1 ]; then
    echo 1
  else
    local prev=$(factorial $((n - 1)))
    echo $((n * prev))
  fi
}

echo "5! = $(factorial 5)"
```

Arrays

Indexed Arrays

bash

```bash
# Array declaration and initialization
fruits=("apple" "banana" "cherry")
numbers=(1 2 3 4 5)

# Alternative declaration
declare -a colors
colors=("red" "green" "blue")

# Adding elements
fruits[3]="date"
fruits+=("elderberry")

# Accessing elements
echo "First fruit: ${fruits[0]}"
echo "All fruits: ${fruits[@]}"
echo "All fruits (quoted): ${fruits[*]}"
echo "Number of fruits: ${#fruits[@]}"

# Array indices
echo "Array indices: ${!fruits[@]}"

# Slicing arrays
echo "First 3 fruits: ${fruits[@]:0:3}"
echo "From index 2: ${fruits[@]:2}"
```

Associative Arrays

bash

```bash
# Declare associative array (Bash 4.0+)
declare -A person
person[name]="John Doe"
person[age]=30
person[city]="New York"

# Alternative initialization
declare -A colors=(
  [red]="#FF0000"
  [green]="#00FF00"
  [blue]="#0000FF"
)

# Accessing values
echo "Name: ${person[name]}"
echo "Age: ${person[age]}"

# Getting all keys and values
echo "All keys: ${!person[@]}"
echo "All values: ${person[@]}"

# Iterating over associative array
for key in "${!person[@]}"; do
  echo "$key: ${person[$key]}"
done
```

Array Operations

bash

```bash
# Copying arrays
original=("a" "b" "c")
copy=("${original[@]}")

# Merging arrays
array1=("1" "2" "3")
array2=("4" "5" "6")
merged=("${array1[@]}" "${array2[@]}")

# Removing elements
unset fruits[1]  # Remove element at index 1
fruits=("${fruits[@]}")  # Re-index array

# Array sorting
numbers=(5 2 8 1 9)
IFS=$'\n' sorted=($(sort <<<"${numbers[*]}")); unset IFS
echo "Sorted: ${sorted[@]}"

# Array searching
search_array() {
  local search_term=$1
  shift
  local array=("$@")

  for i in "${!array[@]}"; do
    if [[ "${array[i]}" == "$search_term" ]]; then
      return $i  # Return index
    fi
  done
  return -1  # Not found
}

fruits=("apple" "banana" "cherry")
if search_array "banana" "${fruits[@]}"; then
  echo "Found banana at index $?"
fi
```

String Manipulation

String Operations

bash

```bash
string="Hello, World!"

# String length
echo "Length: ${#string}"

# Substring extraction
echo "Substring: ${string:7:5}"   # Extract "World"
echo "From position 7: ${string:7}"
echo "Last 6 characters: ${string: -6}"

# String replacement
echo "${string/World/Universe}"     # Replace first occurrence
echo "${string//l/L}"           # Replace all occurrences
echo "${string/Hello/Hi}"        # Replace "Hello" with "Hi"
echo "${string/#Hello/Hi}"        # Replace at beginning
echo "${string/%World!/Universe!}"  # Replace at end

# Case conversion
echo "${string,,}"  # Lowercase
echo "${string^^}"  # Uppercase
echo "${string^}"   # Capitalize first letter
echo "${string,}"   # Lowercase first letter
```

Pattern Matching and Validation

bash

```bash
# Check if string matches pattern
check_email() {
  local email=$1
  if [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "Valid email"
    return 0
  else
    echo "Invalid email"
    return 1
  fi
}

check_email "user@example.com"
check_email "invalid-email"

# Phone number validation
check_phone() {
  local phone=$1
  if [[ $phone =~ ^[0-9]{3}-[0-9]{3}-[0-9]{4}$ ]]; then
    echo "Valid phone number"
  else
    echo "Invalid phone number format (use XXX-XXX-XXXX)"
  fi
}

check_phone "123-456-7890"
```

String Processing

bash

```bash
# Split string into array
IFS=',' read -ra parts <<< "apple,banana,cherry"
for part in "${parts[@]}"; do
  echo "Part: $part"
done

# Join array into string
fruits=("apple" "banana" "cherry")
IFS=', '; joined="${fruits[*]}"; IFS=' '
echo "Joined: $joined"

# Remove whitespace
trim() {
  local string=$1
  string="${string#"${string%%[![:space:]]*}"}"  # Remove leading
  string="${string%"${string##*[![:space:]]}"}"  # Remove trailing
  echo "$string"
}

result=$(trim "  hello world  ")
echo "Trimmed: '$result'"

# String padding
pad_left() {
  local string=$1
  local width=$2
  local char=${3:-' '}
  printf "%*s" $width "$string" | tr ' ' "$char"
}

echo "Padded: '$(pad_left "hello" 10 "0")'"
```

File Operations

File Testing and Information

bash

```bash
file="/path/to/file.txt"

# File existence and type tests
if [[ -e "$file" ]]; then
  echo "File exists"
fi

if [[ -f "$file" ]]; then
  echo "Is a regular file"
elif [[ -d "$file" ]]; then
  echo "Is a directory"
elif [[ -L "$file" ]]; then
  echo "Is a symbolic link"
fi

# Permission tests
[[ -r "$file" ]] && echo "Readable"
[[ -w "$file" ]] && echo "Writable"
[[ -x "$file" ]] && echo "Executable"

# File size and modification
if [[ -s "$file" ]]; then
  echo "File is not empty"
  echo "Size: $(stat -f%z "$file" 2>/dev/null || stat -c%s "$file" 2>/dev/null) bytes"
fi

# File age
if [[ "$file" -nt "/tmp/reference" ]]; then
  echo "File is newer than reference"
fi
```

Reading Files

```bash
```

```bash
# Read entire file into variable
content=$(cat "file.txt")
echo "$content"

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

# Read file with line numbers
line_num=1
while IFS= read -r line; do
    echo "$line_num: $line"
    ((line_num++))
done < "file.txt"

# Process CSV file
while IFS=',' read -r name age city; do
    echo "Name: $name, Age: $age, City: $city"
done < "data.csv"

# Read file into array
mapfile -t lines < "file.txt"
# or
readarray -t lines < "file.txt"

for line in "${lines[@]}"; do
    echo "Line: $line"
done
```

Writing Files

bash

```bash
# Write to file (overwrite)
echo "Hello, World!" > "output.txt"

# Append to file
echo "Another line" >> "output.txt"

# Write multiple lines
cat > "multi_line.txt" << EOF
Line 1
Line 2
Line 3
EOF

# Write array to file
data=("apple" "banana" "cherry")
printf "%s\n" "${data[@]}" > "fruits.txt"

# Write with formatting
printf "Name: %-10s Age: %3d\n" "John" 30 > "formatted.txt"
```

Directory Operations

bash

```bash
# Create directory
mkdir -p "/path/to/nested/directory"

# Remove directory
rmdir "empty_directory"
rm -rf "directory_with_contents"

# List directory contents
for item in /path/to/directory/*; do
  if [[ -f "$item" ]]; then
    echo "File: $(basename "$item")"
  elif [[ -d "$item" ]]; then
    echo "Directory: $(basename "$item")"
  fi
done

# Find files
find . -name "*.txt" -type f
find . -name "*.log" -mtime -7  # Modified in last 7 days
find . -size +1M            # Larger than 1MB

# Directory traversal
traverse_directory() {
  local dir=$1
  for item in "$dir"/*; do
    if [[ -d "$item" ]]; then
      echo "Directory: $item"
      traverse_directory "$item"
    elif [[ -f "$item" ]]; then
      echo "File: $item"
    fi
  done
}

traverse_directory "/path/to/start"
```

Process Management

Running Commands

bash

```bash
# Run command and capture output
output=$(ls -la)
echo "$output"

# Run command and capture both stdout and stderr
output=$(command 2>&1)

# Run command in background
long_running_command &
bg_pid=$!
echo "Started background process with PID: $bg_pid"

# Wait for background process
wait $bg_pid
echo "Background process completed with exit code: $?"

# Run multiple commands in background
{
  command1
  command2
  command3
} &

# Check if process is running
if ps -p $bg_pid > /dev/null 2>&1; then
  echo "Process is still running"
else
  echo "Process has finished"
fi
```

Process Control

bash

```bash
# Kill process by PID
kill_process() {
  local pid=$1
  if ps -p $pid > /dev/null 2>&1; then
    echo "Killing process $pid"
    kill $pid

    # Wait a bit and force kill if necessary
    sleep 2
    if ps -p $pid > /dev/null 2>&1; then
      echo "Force killing process $pid"
      kill -9 $pid
    fi
  else
    echo "Process $pid is not running"
  fi
}

# Find and kill process by name
killall_by_name() {
  local process_name=$1
  local pids=$(pgrep "$process_name")

  if [[ -n "$pids" ]]; then
    echo "Killing processes: $pids"
    kill $pids
  else
    echo "No processes found with name: $process_name"
  fi
}

# Monitor process
monitor_process() {
  local pid=$1
  while ps -p $pid > /dev/null 2>&1; do
    echo "Process $pid is running..."
    sleep 5
  done
  echo "Process $pid has finished"
}
```

Job Control

```bash
# Start job in background
sleep 30 &
job1_pid=$!

# List jobs
jobs

# Bring job to foreground
# fg %1

# Send job to background
# bg %1

# Kill job
kill %1

# Wait for all background jobs
wait

# Trap signals
cleanup() {
  echo "Cleaning up..."
  # Kill background processes
  jobs -p | xargs -r kill
  exit 0
}

trap cleanup SIGINT SIGTERM
```

Error Handling

Exit Codes and Error Checking

```bash
```

```bash
# Check exit code of last command
if command; then
  echo "Command succeeded"
else
  echo "Command failed with exit code: $?"
fi

# Alternative syntax
command && echo "Success" || echo "Failed"

# Set strict error handling
set -e  # Exit on any non-zero exit code
set -u  # Exit on undefined variables
set -o pipefail  # Exit on pipe failures

# Custom exit codes
exit_with_error() {
  local message=$1
  local exit_code=${2:-1}
  echo "Error: $message" >&2
  exit $exit_code
}

# Validate required arguments
if [[ $# -lt 2 ]]; then
  exit_with_error "Usage: $0 <arg1> <arg2>" 2
fi
```

Try-Catch Pattern

bash

```bash
# Simulate try-catch with functions
try() {
    local exit_code=0
    "$@" || exit_code=$?
    return $exit_code
}

catch() {
    local exit_code=$1
    shift
    if [[ $exit_code -ne 0 ]]; then
        "$@"
        return $exit_code
    fi
}

# Usage example
try risky_command && {
    echo "Command succeeded"
} || catch $? {
    echo "Command failed with exit code: $?"
    # Handle error
}
```

Logging and Debugging

bash

```bash
# Logging function
log() {
  local level=$1
  shift
  local message="$*"
  local timestamp=$(date '+%Y-%m-%d %H:%M:%S')
  echo "[$timestamp] [$level] $message" | tee -a script.log
}

log "INFO" "Script started"
log "ERROR" "Something went wrong"
log "DEBUG" "Variable value: $variable"

# Debug function
debug() {
  if [[ "${DEBUG:-0}" -eq 1 ]]; then
    echo "DEBUG: $*" >&2
  fi
}

# Enable debugging
export DEBUG=1
debug "This will be shown"

# Function to print line numbers (for debugging)
print_line() {
  echo "Line ${BASH_LINENO[0]}: $*"
}
```

Signal Handling

bash

```bash
# Cleanup function
cleanup() {
  echo "Performing cleanup..."
  # Remove temporary files
  rm -f /tmp/script_temp_*
  # Kill background processes
  jobs -p | xargs -r kill
  echo "Cleanup completed"
  exit 0
}

# Trap signals
trap cleanup SIGINT SIGTERM SIGQUIT

# Ignore certain signals
trap '' SIGUSR1

# Reset signal handling
trap - SIGINT

# Handle errors with trap
error_handler() {
  local line_number=$1
  echo "Error occurred in script at line: $line_number"
  exit 1
}

trap 'error_handler ${LINENO}' ERR
```

Advanced Features

Command Line Argument Processing

bash

```
# Using getopts for option parsing
usage() {
  echo "Usage: $0 [-v] [-f file] [-h]"
  echo "  -v       Verbose mode"
  echo "  -f file    Input file"
  echo "  -h       Show help"
  exit 1
}

verbose=false
input_file=""

while getopts "vf:h" opt; do
  case $opt in
    v)
      verbose=true
      ;;
    f)
      input_file="$OPTARG"
      ;;
    h)
      usage
      ;;
    \?)
      echo "Invalid option: -$OPTARG" >&2
      usage
      ;;
    :)
      echo "Option -$OPTARG requires an argument" >&2
      usage
      ;;
  esac
done

# Shift to remove processed options
shift $((OPTIND-1))

# Remaining arguments are in $@
if [[ $verbose == true ]]; then
  echo "Verbose mode enabled"
fi

if [[ -n "$input_file" ]]; then
```

```
    echo "Input file: $input_file"
fi

echo "Remaining arguments: $@"
```

Advanced Parameter Processing

bash

```
# Long option support
parse_args() {
  while [[ $# -gt 0 ]]; do
    case $1 in
      --verbose|-v)
        VERBOSE=true
        shift
        ;;
      --file=*)
        INPUT_FILE="${1#*=}"
        shift
        ;;
      --file|-f)
        INPUT_FILE="$2"
        shift 2
        ;;
      --output|-o)
        OUTPUT_FILE="$2"
        shift 2
        ;;
      --help|-h)
        show_help
        exit 0
        ;;
      --)
        shift
        break
        ;;
      -*)
        echo "Unknown option: $1" >&2
        exit 1
        ;;
      *)
        POSITIONAL_ARGS+=("$1")
        shift
        ;;
    esac
  done
}

# Configuration file support
load_config() {
  local config_file=${1:-"$HOME/.scriptrc"}
```

```
    if [[ -f "$config_file" ]]; then
        source "$config_file"
    fi
}

# Default values
VERBOSE=false
INPUT_FILE=""
OUTPUT_FILE=""
POSITIONAL_ARGS=()

load_config
parse_args "$@"
```

Regular Expressions

```bash
```

```bash
# Pattern matching with regex
validate_input() {
    local input=$1

    # Email validation
    if [[ $input =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
        echo "Valid email"
        return 0
    fi

    # Phone number validation
    if [[ $input =~ ^(\+?1-?)?[0-9]{3}-?[0-9]{3}-?[0-9]{4}$ ]]; then
        echo "Valid phone number"
        return 0
    fi

    # IP address validation
    if [[ $input =~ ^([0-9]{1,3}\.){3}[0-9]{1,3}$ ]]; then
        echo "Valid IP address"
        return 0
    fi

    echo "Invalid input format"
    return 1
}

# Extract information using regex
extract_info() {
    local text="Contact John Doe at john@example.com or call 555-123-4567"

    # Extract email
    if [[ $text =~ ([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}) ]]; then
        echo "Email: ${BASH_REMATCH[1]}"
    fi

    # Extract phone number
    if [[ $text =~ ([0-9]{3}-[0-9]{3}-[0-9]{4}) ]]; then
        echo "Phone: ${BASH_REMATCH[1]}"
    fi
}
```

Working with JSON

bash

```bash
# Parse JSON using jq (if available)
parse_json() {
  local json_file=$1

  if command -v jq >/dev/null 2>&1; then
    # Extract specific fields
    name=$(jq -r '.name' "$json_file")
    age=$(jq -r '.age' "$json_file")
    echo "Name: $name, Age: $age"

    # Extract array elements
    jq -r '.hobbies[]' "$json_file" | while read -r hobby; do
      echo "Hobby: $hobby"
    done
  else
    echo "jq not available, using basic parsing"
    # Basic JSON parsing without jq
    grep -o '"name":"[^"]*"' "$json_file" | cut -d'"' -f4
  fi
}

# Create JSON output
create_json() {
  local name=$1
  local age=$2

  cat << EOF
{
  "name": "$name",
  "age": $age,
  "timestamp": "$(date -u +%Y-%m-%dT%H:%M:%SZ)",
  "system": "$(uname -s)"
}
EOF
}
```

Working with APIs

bash

```
# Make HTTP requests
api_request() {
  local method=$1
  local url=$2
  local data=$3

  if command -v curl >/dev/null 2>&1; then
    case $method in
      GET)
        curl -s "$url"
        ;;
      POST)
        curl -s -X POST -H "Content-Type: application/json" -d "$data" "$url"
        ;;
      PUT)
        curl -s -X PUT -H "Content-Type: application/json" -d "$data" "$url"
        ;;
      DELETE)
        curl -s -X DELETE "$url"
        ;;
    esac
  else
    echo "curl not available"
    return 1
  fi
}

# Example usage
response=$(api_request GET "https://api.github.com/users/octocat")
echo "$response" | jq -r '.name'
```

Best Practices

Script Organization

```
bash
```

```bash
#!/bin/bash

# Script: example_script.sh
# Purpose: Demonstrate best practices
# Author: Your Name
# Version: 1.0
# Created: $(date)

# Strict error handling
set -euo pipefail

# Global variables
readonly SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
readonly SCRIPT_NAME="$(basename "$0")"
readonly LOG_FILE="/var/log/${SCRIPT_NAME%.sh}.log"

# Color codes
readonly RED='\033[0;31m'
readonly GREEN='\033[0;32m'
readonly YELLOW='\033[1;33m'
readonly NC='\033[0m'

# Configuration
DEBUG=${DEBUG:-0}
VERBOSE=${VERBOSE:-0}

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

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

  case $level in
    ERROR)
      echo -e "${RED}ERROR: $message${NC}" >&2
      ;;
    WARN)
      echo -e "${YELLOW}WARNING: $message${NC}" >&2
      ;;
    INFO)
```

```
        [[ $VERBOSE -eq 1 ]] && echo -e "${GREEN}INFO: $message${NC}"
        ;;
      DEBUG)
        [[ $DEBUG -eq 1 ]] && echo -e "DEBUG: $message" >&2
        ;;
  esac
}

# Main function
main() {
  log "INFO" "Script started"

  # Your main logic here

  log "INFO" "Script completed successfully"
}

# Cleanup function
cleanup() {
  log "INFO" "Performing cleanup"
  # Cleanup code here
}

# Signal handling
trap cleanup SIGINT SIGTERM

# Parameter validation
if [[ $# -lt 1 ]]; then
  echo "Usage: $0 <argument>" >&2
  exit 1
fi

# Run main function
main "$@"
```

Code Quality Guidelines

bash

```bash
# Use meaningful variable names
user_name="john_doe"       # Good
un="john_doe"              # Bad

# Use constants for fixed values
readonly MAX_RETRIES=3
readonly DEFAULT_TIMEOUT=30

# Quote variables to prevent word splitting
echo "$user_name"          # Good
echo $user_name            # Bad (can break with spaces)

# Use [[ ]] instead of [ ] for conditions
if [[ -f "$file" ]]; then    # Good
  echo "File exists"
fi

if [ -f $file ]; then        # Bad (unquoted variable)
  echo "File exists"
fi

# Use printf instead of echo for formatted output
printf "User: %-10s Age: %3d\n" "$name" "$age"  # Good
echo "User: $name Age: $age"                    # Basic

# Use local variables in functions
process_user() {
  local name=$1          # Good
  local age=$2
  # Process user
}

process_user() {
  name=$1                # Bad (global variable)
  age=$2
  # Process user
}

# Check command existence before using
if command -v git >/dev/null 2>&1; then
  git status
else
  echo "Git is not installed"
```

```
fi

# Use arrays for lists
files=("file1.txt" "file2.txt" "file3.txt")  # Good
files="file1.txt file2.txt file3.txt"       # Bad
```

Security Best Practices

bash

```bash
# Validate input
validate_filename() {
    local filename=$1

    # Check for null bytes
    if [[ "$filename" == *\0'* ]]; then
        return 1
    fi

    # Check for path traversal
    if [[ "$filename" == *..* ]]; then
        return 1
    fi

    # Check length
    if [[ ${#filename} -gt 255 ]]; then
        return 1
    fi

    return 0
}

# Secure temporary files
create_temp_file() {
    local temp_file
    temp_file=$(mktemp) || {
        echo "Failed to create temporary file" >&2
        return 1
    }

    # Set restrictive permissions
    chmod 600 "$temp_file"

    echo "$temp_file"
}

# Clean up temporary files
cleanup_temp() {
    if [[ -n "${temp_files:-}" ]]; then
        rm -f "${temp_files[@]}"
    fi
}
```

```
# Use parameter expansion instead of eval
# Good
value=${!var_name}

# Bad - security risk
# eval "value=|$var_name"
```

Common Patterns

Configuration Management

bash

```
# Load configuration from file
load_config() {
    local config_file=$1

    if [[ ! -f "$config_file" ]]; then
        echo "Config file not found: $config_file" >&2
        return 1
    fi

    # Source config file safely
    if source "$config_file" 2>/dev/null; then
        log "INFO" "Configuration loaded from $config_file"
    else
        log "ERROR" "Failed to load configuration from $config_file"
        return 1
    fi
}

# Configuration with defaults
set_defaults() {
    # Server configuration
    SERVER_HOST=${SERVER_HOST:-"localhost"}
    SERVER_PORT=${SERVER_PORT:-8080}

    # Database configuration
    DB_HOST=${DB_HOST:-"localhost"}
    DB_PORT=${DB_PORT:-5432}
    DB_NAME=${DB_NAME:-"myapp"}

    # Application settings
    LOG_LEVEL=${LOG_LEVEL:-"INFO"}
    MAX_WORKERS=${MAX_WORKERS:-4}
}
```

Backup and Archive Scripts

bash

```bash
# Backup function with rotation
backup_files() {
  local source_dir=$1
  local backup_dir=$2
  local max_backups=${3:-7}

  local timestamp=$(date +%Y%m%d_%H%M%S)
  local backup_name="backup_${timestamp}.tar.gz"
  local backup_path="${backup_dir}/${backup_name}"

  # Create backup directory if it doesn't exist
  mkdir -p "$backup_dir"

  # Create backup
  log "INFO" "Creating backup: $backup_path"
  if tar -czf "$backup_path" -C "$(dirname "$source_dir")" "$(basename "$source_dir")"; then
    log "INFO" "Backup created successfully"
  else
    log "ERROR" "Failed to create backup"
    return 1
  fi

  # Rotate old backups
  local backup_count=$(ls -1 "$backup_dir"/backup_*.tar.gz 2>/dev/null | wc -l)
  if [[ $backup_count -gt $max_backups ]]; then
    log "INFO" "Removing old backups (keeping $max_backups)"
    ls -1t "$backup_dir"/backup_*.tar.gz | tail -n +$((max_backups + 1)) | xargs rm -f
  fi
}

# Usage
backup_files "/home/user/documents" "/backups" 5
```

Service Management Script

bash

```
# Service management template
SERVICE_NAME="myapp"
PID_FILE="/var/run/${SERVICE_NAME}.pid"
LOG_FILE="/var/log/${SERVICE_NAME}.log"
SERVICE_USER="myapp"

start_service() {
    if is_running; then
        echo "Service is already running"
        return 1
    fi

    echo "Starting $SERVICE_NAME..."
    # Start your service here
    # sudo -u $SERVICE_USER /path/to/your/service &
    # echo $! > $PID_FILE

    sleep 2
    if is_running; then
        echo "$SERVICE_NAME started successfully"
    else
        echo "Failed to start $SERVICE_NAME"
        return 1
    fi
}

stop_service() {
    if ! is_running; then
        echo "Service is not running"
        return 1
    fi

    local pid=$(cat "$PID_FILE")
    echo "Stopping $SERVICE_NAME (PID: $pid)..."

    kill "$pid"

    # Wait for graceful shutdown
    local count=0
    while is_running && [[ $count -lt 30 ]]; do
        sleep 1
        ((count++))
    done
```

```
    if is_running; then
        echo "Force killing $SERVICE_NAME"
        kill -9 "$pid"
    fi

    rm -f "$PID_FILE"
    echo "$SERVICE_NAME stopped"
}

is_running() {
    [[ -f "$PID_FILE" ]] && ps -p "$(cat "$PID_FILE")" >/dev/null 2>&1
}

status_service() {
    if is_running; then
        local pid=$(cat "$PID_FILE")
        echo "$SERVICE_NAME is running (PID: $pid)"
    else
        echo "$SERVICE_NAME is not running"
    fi
}

case "$1" in
    start)
        start_service
        ;;
    stop)
        stop_service
        ;;
    restart)
        stop_service
        sleep 2
        start_service
        ;;
    status)
        status_service
        ;;
    *)
        echo "Usage: $0 {start|stop|restart|status}"
        exit 1
        ;;
esac
```

Log Analysis Script

bash

```
# Log analysis functions
analyze_logs() {
    local log_file=$1
    local time_window=${2:-"1 hour ago"}

    echo "=== Log Analysis Report ==="
    echo "File: $log_file"
    echo "Time window: $time_window"
    echo "Analysis date: $(date)"
    echo

    # Get logs from specified time window
    local since_timestamp=$(date -d "$time_window" '+%Y-%m-%d %H:%M:%S')

    # Error count
    local error_count=$(grep -c "ERROR" "$log_file" || echo "0")
    echo "Total errors: $error_count"

    # Warning count
    local warning_count=$(grep -c "WARN" "$log_file" || echo "0")
    echo "Total warnings: $warning_count"

    # Most common errors
    echo
    echo "=== Most Common Errors ==="
    grep "ERROR" "$log_file" | awk '{print $NF}' | sort | uniq -c | sort -nr | head -5

    # Traffic analysis
    echo
    echo "=== Traffic Analysis ==="
    grep -E "GET|POST|PUT|DELETE" "$log_file" | awk '{print $7}' | sort | uniq -c | sort -nr | head -10
}

# Real-time log monitoring
monitor_logs() {
    local log_file=$1
    local pattern=${2:-"ERROR|WARN"}

    echo "Monitoring $log_file for pattern: $pattern"
    echo "Press Ctrl+C to stop"

    tail -f "$log_file" | grep --line-buffered -E "$pattern" | while read -r line; do
        local timestamp=$(date '+%Y-%m-%d %H:%M:%S')
```

```
    echo "[$timestamp] $line"

    # Send alert for critical errors
    if echo "$line" | grep -q "CRITICAL"; then
      # Send notification (email, Slack, etc.)
      echo "ALERT: Critical error detected" | mail -s "Log Alert" admin@example.com
    fi
  done
}
```

Debugging and Testing

Debugging Techniques

bash

```bash
# Debug mode
if [[ "${DEBUG:-0}" -eq 1 ]]; then
  set -x  # Print commands as they execute
fi

# Debug function
debug() {
  if [[ "${DEBUG:-0}" -eq 1 ]]; then
    echo "DEBUG: $*" >&2
  fi
}

# Trace function execution
trace() {
  echo "TRACE: Entering function ${FUNCNAME[1]}" >&2
  echo "TRACE: Arguments: $*" >&2
}

# Example function with tracing
process_data() {
  trace "$@"
  local data=$1
  debug "Processing data: $data"

  # Function logic here

  debug "Data processing completed"
}

# Line number debugging
debug_line() {
  echo "DEBUG: Line ${BASH_LINENO[0]} in ${BASH_SOURCE[1]}: $*" >&2
}

# Variable debugging
debug_var() {
  local var_name=$1
  local var_value=${!var_name}
  echo "DEBUG: $var_name = '$var_value'" >&2
}

# Usage
```

```
name="John"
debug_var "name"
```

Testing Framework

bash

```
# Simple testing framework
TEST_COUNT=0
TEST_PASSED=0
TEST_FAILED=0

assert_equals() {
  local expected=$1
  local actual=$2
  local message=${3:-"Assertion failed"}

  ((TEST_COUNT++))

  if [[ "$expected" == "$actual" ]]; then
    echo "✓ PASS: $message"
    ((TEST_PASSED++))
  else
    echo "✗ FAIL: $message"
    echo "  Expected: '$expected'"
    echo "  Actual: '$actual'"
    ((TEST_FAILED++))
  fi
}

assert_true() {
  local condition=$1
  local message=${2:-"Assertion failed"}

  if $condition; then
    assert_equals "true" "true" "$message"
  else
    assert_equals "true" "false" "$message"
  fi
}

run_tests() {
  echo "Running tests..."

  # Test string manipulation
  local result=$(echo "hello" | tr '[:lower:]' '[:upper:]')
  assert_equals "HELLO" "$result" "String to uppercase"

  # Test file operations
  touch /tmp/test_file
```

```
    assert_true "[[ -f /tmp/test_file ]]" "File creation test"
    rm -f /tmp/test_file

    # Test arithmetic
    local sum=$((5 + 3))
    assert_equals "8" "$sum" "Addition test"

    # Print results
    echo
    echo "Test Results:"
    echo "  Total: $TEST_COUNT"
    echo "  Passed: $TEST_PASSED"
    echo "  Failed: $TEST_FAILED"

    if [[ $TEST_FAILED -eq 0 ]]; then
        echo "All tests passed!"
        return 0
    else
        echo "Some tests failed!"
        return 1
    fi
}

# Run tests if script is executed directly
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
    run_tests
fi
```

Performance Monitoring

bash

```bash
# Timing functions
time_function() {
    local func_name=$1
    shift

    local start_time=$(date +%s.%N)
    "$func_name" "$@"
    local exit_code=$?
    local end_time=$(date +%s.%N)

    local duration=$(echo "$end_time - $start_time" | bc)
    echo "Function '$func_name' took $duration seconds" >&2

    return $exit_code
}

# Memory usage monitoring
monitor_memory() {
    local pid=${1:-$}

    while ps -p $pid >/dev/null 2>&1; do
        local memory=$(ps -o rss= -p $pid)
        echo "Memory usage: ${memory}KB"
        sleep 1
    done
}

# Resource usage summary
show_resource_usage() {
    echo "=== Resource Usage Summary ==="
    echo "CPU usage: $(top -l 1 | awk '/CPU usage/ {print $3}' | cut -d% -f1)%"
    echo "Memory usage: $(free | awk '/^Mem:/ {printf "%.2f%%", $3/$2 * 100.0}' 2>/dev/null || echo "N/A")"
    echo "Disk usage: $(df -h . | awk 'NR==2 {print $5}')"
    echo "Load average: $(uptime | awk -F'load average:' '{print $2}')"
}
```

Performance Optimization

Efficient Coding Patterns

bash

```
# Use parameter expansion instead of external commands
# Slow
basename=$(basename "$path")
dirname=$(dirname "$path")

# Fast
basename=${path##*/}
dirname=${path%/*}

# Use built-in string operations instead of external tools
# Slow
length=$(echo "$string" | wc -c)
upper=$(echo "$string" | tr '[:lower:]' '[:upper:]')

# Fast
length=${#string}
upper=${string^^}

# Avoid unnecessary subprocesses
# Slow
if [ $(echo "$string" | grep -c "pattern") -gt 0 ]; then
  echo "Pattern found"
fi

# Fast
if [[ "$string" == *"pattern"* ]]; then
  echo "Pattern found"
fi

# Use arrays for collections instead of strings
# Slow
files="file1.txt file2.txt file3.txt"
for file in $files; do
  echo "$file"
done

# Fast
files=("file1.txt" "file2.txt" "file3.txt")
for file in "${files[@]}"; do
  echo "$file"
done
```

Optimizing Loops

bash

```
# Pre-calculate loop conditions
# Slow
for ((i=0; i<$(wc -l < file.txt); i++)); do
  # Loop body
done

# Fast
line_count=$(wc -l < file.txt)
for ((i=0; i<line_count; i++)); do
  # Loop body
done

# Use built-in read instead of external commands
# Slow
cat file.txt | while read line; do
  echo "$line"
done

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

# Batch operations when possible
# Slow
for file in *.txt; do
  chmod 644 "$file"
done

# Fast
chmod 644 *.txt
```

Memory Optimization

bash

```bash
# Avoid loading large files into memory
# Memory-intensive
content=$(cat large_file.txt)
process_content "$content"

# Memory-efficient
process_file_streaming() {
  while IFS= read -r line; do
    # Process line by line
    echo "Processing: $line"
  done < large_file.txt
}

# Use local variables to free memory
process_data() {
  local large_array=()
  # Populate array
  # Process array
  # Array is automatically freed when function exits
}

# Clear variables when no longer needed
large_variable="lots of data"
# Use variable
unset large_variable
```

Quick Reference

Common Commands Cheat Sheet

bash

```
# Variable operations
${var:-default}      # Use default if var is unset
${var:=default}      # Set var to default if unset
${var:+alternate}    # Use alternate if var is set
${#var}              # Length of var
${var%pattern}       # Remove pattern from end
${var#pattern}       # Remove pattern from beginning

# Conditional expressions
[[ -e file ]]        # File exists
[[ -f file ]]        # Is regular file
[[ -d file ]]        # Is directory
[[ -r file ]]        # Is readable
[[ -w file ]]        # Is writable
[[ -x file ]]        # Is executable
[[ str =~ pattern ]] # Regex match
[[ str == pattern ]] # Pattern match

# Process management
command &            # Run in background
wait $!              # Wait for last background job
jobs                 # List active jobs
kill %1              # Kill job 1
nohup command &      # Run immune to hangups

# I/O redirection
> file               # Redirect stdout to file
>> file              # Append stdout to file
2> file              # Redirect stderr to file
&> file              # Redirect both stdout and stderr
< file               # Redirect file to stdin
<<< "string"         # Here string
```

This comprehensive guide covers the essential aspects of Bash scripting with practical examples and detailed explanations. Use it as a reference for writing robust, maintainable shell scripts.