Bash Programming

Kameswari Chebrolu

Introduction

- Command-line bash vs bash script
- Scripting: take a set of commands you'd normally run by hand and put them in a file
  Why?
  - Automate tasks to make life easier!
  - Can run them repeatedly with one command!

Bash Script Example

```
#!/bin/bash

mkdir demo
cd demo
mkdir code
mkdir doc
cd code
cp ../../hello.c ./
gcc -o hello hello.c
./hello
```

Explanation

- Script made of a sequence of commands
- Two ways to run the script:
  - Invoke a script using bash
  - e.g. bash 00-bash.sh

- Invoke a script as a hash-bang executable
  - Script contains #!/usr/bin/env bash or #!/bin/bash (absolute path)
    - env will find the path, better to use in practice!
  - Above is called shebang
    - Sharp" (#) + "Bang" (!): # is called "sharp" in programming and ! is called "bang" in unix
    - A line of code that tells the shell what program to use
    - # is used for commenting, hence ignored during execution
  - Want to use python, replace bash with python above
  - You need to make the file executable via chmod
    - chmod +x 00-shebang.sh (changes permission)
    - ./00-shebang.sh (runs it)

Print

- To print messages and debug Bash scripts, echo is a simple and effective command
  - E.g. echo "Hello, World!"
- Can use echo to enable escape sequences, print to files, print in color etc

```
#!/bin/bash

echo "Hello, World!"

#ignores special characters
echo "Line1\nLine2\tTabbed"
#-e enable escape sequences, so newline and tab will be printed
echo -e "Line1\nLine2\tTabbed"

#Can print to files also
echo "Starting script..." >> debug.log

#printing in color
#\e[32m - ANSI Escape Code for Green Text; \e → Escape character; [ → Starts the ANSI sequence.
#32m → Sets the text color to green; 34m is blue, 31m is red etc
#\e[0m - Resets Formatting
echo -e "\e[32mSuccess: Task completed!\e[0m"
echo -e "\e[31mError: Something went wrong!\e[0m"

#How echo handles spaces
echo word1       word2
echo "word1      word2"
```

Variables

- No type casting, no need for declaration
  - Bash variables are character strings, but, depending on context permits integer operations

- Defined using the assign operator "="
  - e.g. age=20
  - No space before or after!
  - Variable name can include alphabets, digits, and underscore
    - Can start with alphabets and underscore only
    - Are case sensitive
  - To use a variable, prefix it with $

Few things to note

- Special character need escaping via single quotes ('...') or back slash (\)
  - \” or \’ or \\ or \$
- Single and double quotes: Help group arguments with spaces
  - Single quotes (') preserve the literal value of each character
  - Double quotes (“”) preserve the literal value of all characters with the exception of $, `, \
    - Variables will be expanded and commands substituted

“ or ‘

Feature	`"Double Quotes"`	`'Single Quotes'`
Variable Expansion	✅ Yes	❌ No
Command Substitution	✅ Yes	❌ No
Special Characters (`\` , `$` , `*`)	✅ Yes (Some)	❌ No (Everything is literal)

```
#!/bin/bash

#name = "Alice"  (Incorrect, spaces are not allowed)
name="Bhargav"    # Correct
age=12          # Correct
# printing and using variables via echo and $
echo "Hello, $name!"
echo "You are $age years old."

#numbers are also strings
echo -e "\n"
num1=1234
num2=7890
echo $num1$num2
```

```
#difference between " and '
echo -e "\n"
myVariable="Hello, world\!"
echo myVariable
echo $myVariable
echo "$myVariable"
echo '$myVariable'

#Handling spaces
#{ } can be used to clearly separate the variable name from surrounding text. See
below
echo -e "\n"
var="abc    xyz"
num="123"
echo 1 $var$num
echo 2 "$varXX$num"
echo 3 "${var}XX${num}"
```

```
#command substitution in variables
echo -e "\n"
lsResult=$(ls)
directory=`pwd`
echo "My files are:" $lsResult  in $directory


#Double quotes (“”) preserve the literal value of all characters with the exception of
$, `, \
#In the first two commands, $ is interpreted as a variable with or  without quotes.
#In the third command, bash intereprets * as wildcard and lists all files ending in
.sh.
#In the fourth, since it is inside quotes, it views it as * character and looks for a
file named "*.sh"
echo -e "\n"
echo $HOME
echo "$HOME"
ls *.sh
ls "*.sh"
```

Environment Variables

- Variables in your system that describe your environment!
  - SHELL: what shell you're running
  - USER: username of the current user
  - PWD: present working directory
  - PATH: specifies the directories to be searched to find a command
  - etc
- "env" command (type in terminal) shows the list of variables

```
#!/bin/bash

#enviornment variables
echo $HOME # Home Directory
echo $PWD # current working directory
echo $BASH # Bash shell name
echo $BASH_VERSION # Bash shell Version
echo $LOGNAME # Name of the Login User
echo $OSTYPE # Type of OS
echo "User $LOGNAME is working in folder $PWD using OS $OSTYPE"
```

Arithmetic Expressions

- Bash provides some basic supports (not great) for arithmetic operations using multiple methods
- let: built-in command for evaluating arithmetic expressions
  - E.g. let “a = 1” or let “a = a + 1”
  - Use let when doing inline arithmetic operations that modify variables

- $[]: older way to evaluate arithmetic expressions (deprecated)
  - E.g. num=$[5 + 3]
- $(()): preferred modern method for arithmetic calculations
  - E.g. num=$((5 + 3))
  - Use $(( )) when you need arithmetic expansion and want to directly assign the result

- Bash does not support floating-point arithmetic natively; use bc (basic calculator) command instead
  - By default, bc performs integer arithmetic; use -l option for float
  - E.g. echo "scale=2; 5 / 2" | bc -l
    - -l loads the math library and sets scale=20 by default
    - Can change scale (precision) to what you want also
- Can use declare -i to make a variable integer-only
  - E.g. declare -i num=5
  - Any non-numeric value assigned to num will be ignored

```
#!/bin/bash

#Cannot use expression directly, will be viewed as a string
echo 1 5*2+1
num=5*2+1
echo 2 $num

#using let
let num=5*2+1
echo 3 $num
#a = 5  (Incorrect because Bash interprets a as a command, = as an argument, and
5 as another argument
#with let, "a = 5" is the argument to let; so inside the quotes spaces are ok
let "a = 5"
let "a++"
echo 4 $a
```

```
#Two formats, $((expression)) and $[expression]
echo 5 $[5*2+1] #avoid -- deprecated
echo 6 $((5*2+1))
a=10
b=2
result=$((a ** b))
echo 7 $result
```

```
#Bash does not support floating point arithmetic natively
echo 8 $((10 % 3))  # Output: 1
#floating point needs use of bc
echo 9 $((3/4)) #Output: 0
echo -n "10 "
echo "3/4" | bc -l #Output: .75000000000000000000
#above two lines can also be replaced like this via command substitution echo "10
$(echo "3/4" | bc -l)"
#example involving use of scale to reduce the precision to 5 digits
echo -n "11 "
echo "scale=5; sqrt(49)" | bc -l    # Output: 7.00000
#bc supports variables also; Variables persist within a single bc session but are not
retained outside
echo -n "12 "
echo "a=10; b=3; a/b" | bc -l    # Output: 3.33333333333333333333
#supports conditional expressions
echo -n "13 "
echo "5 > 2" | bc     # Output: 1
```

```
#declare makes variable result integer only
declare -i num
num=$((5/2+1))
echo 14 $num # Ouptput 3
#Any non-numeric value assigned to num will be ignored
num="hello"
echo 15 $num  # Output: 0
```

Operators

- Bash supports a variety of operators for arithmetic, string, logical comparisons, and file testing
- Arithmetic Operators ⟶ +, -, *, /, %, **
- Comparison Operators ⟶ -eq, -ne, -gt, -lt, -ge, -le
- Logical Operators ⟶ &&, ||, !
- String Operators ⟶ =, !=, -z, -n
- File Operators ⟶ -e, -f, -d, -r, -w, -x
- Bitwise Operators ⟶ &, |, ^, ~, <<, >>
- Assignment Operators ⟶ =, +=, -=, *=, /=, %=

```
#!/bin/bash
#Press \U1F539 shows the blue diamond
echo -e " \n \U1F539 Arithmetic Operators"
a=10
b=3
echo "Addition: $((a + b))"
echo "Subtraction: $((a - b))"
echo "Multiplication: $((a * b))"
echo "Division (integer): $((a / b))"
echo "Modulus (remainder): $((a % b))"
echo "Exponentiation: $((a ** b))"
echo "Floating-point division using bc: $(echo "scale=2; $a / $b"| bc -l)"
```

```
echo -e "\n🔹 Comparison Operators"
#&& Runs if True, || Runs if False
[[ $a -eq $b ]] && echo "a is equal to b" || echo "a is not equal to b"
[[ $a -ne $b ]] && echo "a is not equal to b"
[[ $a -gt $b ]] && echo "a is greater than b"
[[ $a -lt $b ]] && echo "a is less than b"
[[ $a -ge 10 ]] && echo "a is greater than or equal to 10"
[[ $b -le 3 ]] && echo "b is less than or equal to 3"

echo -e "\n🔹 Logical Operators"
[[ $a -gt 5 && $b -lt 30 ]] && echo "Both conditions met (AND)"
[[ $a -gt 15 || $b -lt 30 ]] && echo "At least one condition met (OR)"
[[ ! $a -eq 10 ]] && echo "Negation: a is not 10" || echo "a is 10"
```

```
echo -e "\n🔹 String Operators"
str1="hello"
str2="world"
[[ $str1 = $str2 ]] && echo "Strings are equal" || echo "Strings are not
equal"
[[ -z $str3 ]] && echo "String is empty"
[[ -n $str1 ]] && echo "String is not empty"

echo -e "\n🔹 File Test Operators"
touch tempfile.txt
[[ -e tempfile.txt ]] && echo "File exists"
[[ -f tempfile.txt ]] && echo "It is a regular file"
[[ -s tempfile.txt ]] && echo "File is not empty" || echo "File is empty"
[[ -r tempfile.txt ]] && echo "File is readable"
[[ -w tempfile.txt ]] && echo "File is writable"
rm tempfile.txt
```

```
echo -e "\n🔹 Assignment Operators"
#Use $((expr)) when you need to return a computed value to assign (e.g., y=$((x+2))).
#Use ((expr)) when you modify variables directly (e.g., ((x+=1))).
x=5
((x+=2))
echo "x += 2: $x"
((x-=3))
echo "x -= 3: $x"
((x*=5))
echo "x *= 5: $x"
((x/=2))
echo "x /= 2: $x"
((x%=3))
echo "x %= 3: $x"
```

```
echo -e "\n🔹 Bitwise Operators"
a=5  # 0101
b=3  # 0011
echo "AND: $((a & b))"  # 0001 = 1
echo "OR: $((a | b))"   # 0111 = 7
echo "XOR: $((a ^ b))"  # 0110 = 6
echo "NOT a: $((~a))"
echo "Right Shift: $((a >> 1))"
echo "Left Shift: $((a << 1))"
```

Conditionals

- Bash supports conditionals for decision-making in scripts
- Supports if, elif, else

if statement evaluates a condition and executes the block of code if the condition is true

```
if [[ CONDITION ]]; then
  # Code to execute if CONDITION is true
fi
```

if-else Statement: If the condition is false, the else block will be executed.

```
if [[ CONDITION ]]; then
  # Code to execute if CONDITION is true
else
  # Code to execute if CONDITION is false
fi
```

if-elif-else Statement: can check multiple conditions using elif (else if)

```
if [[ CONDITION1 ]]; then
  # Code for CONDITION1
elif [[ CONDITION2 ]]; then
  # Code for CONDITION2
else
  # Code if none of the conditions are true
fi
```

Note: [[...]] is a more modern and powerful alternative to [...] which you may seem sometimes. If you are sure you are working with Bash, you should always prefer [[...]] for conditional expressions!

Feature	`(( ))`	`[[ ]]`
Purpose	Arithmetic evaluation	String and conditional testing
Usage	Math operations and comparisons	String comparisons, regex
Variables	No `$` needed (e.g., `((x++))`)	`$` needed for variables (e.g., `[[ $x -gt 5 ]]`)
String Comparison	Not supported	Supports string comparison and regex (`==`, `!=`, `=~`)
Logical Operators	Supports only basic arithmetic comparison (e.g., `==`, `!=`, `>` and `<`)	Supports logical operators like `&&` (AND), `
Return value	Numeric result (0 or non-zero)	Boolean (true or false)

```
#!/bin/bash
x=10
# Basic if
if (( x > 5 )); then
 echo "1: x is greater than 5"
fi
# If-else
if (( x % 2 == 0 )); then
 echo "2: x is even"
else
 echo "2: x is odd"
fi
# If-elif-else
if (( x > 15 )); then
 echo "3: x is greater than 15"
elif (( x > 5 )); then
 echo "3: x is greater than 5 but less than or equal to 15"
else
 echo "3: x is less than or equal to 5"
fi
```

```
# String comparison with [[ ... ]] as opposed to using (( )) which is for arithmetic
name="Arun"
if [[ "$name" == "Sunita" ]]; then
 echo "4: Hello, Sunita!"
else
 echo "4: Hello, Stranger!"
fi


# Conditional involving commands (modernized with [[ ]] for string comparison)
if ls nonexistent_directory &>/dev/null; then
 echo "6: Directory exists!"
else
 echo "6: Error: Directory does not exist!"
fi
```

Loops

Bash provides three main types of loops for repeating tasks

- for loop – Iterates over a list of values or a range
- while loop – Repeats as long as a condition is true
- until loop – Repeats until a condition becomes true

```bash
#!/usr/bin/env bash

#For loop
echo -e "\n🔹 For Loop (Iterating Over a List of Strings)"
for fruit in apple banana cherry; do
 echo "Fruit: $fruit"
done

echo -e "\n🔹 For Loop (Iterating Over a Numeric Range)"
for i in {1..5}; do
 echo "Number: $i"
done

#for loop with conditional
echo -e "\n🔹 For Loop  with conditional"
for dir in code doc anotherdir
do
   if [ -d "$dir" ]; then
       echo "$dir already exists"
   else
       mkdir "$dir"
   fi
done
```

```
#using command substitution
echo -e "\n🔹 For Loop (Iterating Over Command Output)"
for file in $(ls); do
 echo "File: $file"
done

#demo of break and continue in aloop
echo -e "\n🔹 For Loop with Break (Stopping at 3)"
for i in {1..5}; do
 if (( i == 3 )); then
   echo "Stopping at 3"
   break
 fi
 echo "Number: $i"
done

echo -e "\n🔹 For Loop with Continue (Skipping 3)"
for i in {1..5}; do
 if (( i == 3 )); then
   echo "Skipping 3"
   continue
 fi
 echo "Number: $i"
done
```

```
#While loop
echo -e "\n🔹 While Loop (Counting Up)"
count=1
while (( count <= 5 )); do
 echo "Count: $count"
 (( count++ ))
done

#Another while loop ; Notice use of [[ ]] and not (( )) since we are dealing with string operations
#${variable:offset} → Extracts a substring from variable, starting at offset
#${word:1} → Extract substring starting at index 1 which is "o"; same as removing first character
word="HELLO"
while [[ $word != "" ]]; do
   echo "$word"
   word=${word:1}  # Remove first character
done

#Until Loop
echo -e "\n🔹 Until Loop (Counting Up)"
count=1
until (( count > 5 )); do
 echo "Count: $count"
 (( count++ ))
done
```

Special Shell Variables

- Predefined variables that provide useful information about the script, arguments, process control, exit statuses etc

- Positional Parameters: Hold the arguments passed to a script or function

$0: Script name
$1, $2, … :Arguments passed to the script
$# Number of arguments
$@ All arguments as separate words (i.e. a set of strings)
$* All arguments as a single string (i.e. one string)

- Exit Status Variables

$?: Exit status of the last command (0 = success, nonzero = failure)

- Process Control Variables

$$: Process ID (PID) of the current shell

- Input & Output Variables

$_: Last argument of the last command

```
#!/bin/bash
#Run this script as follows: bash 08-shell-variables.sh abc "de fg" hij
# Note it has 3 arguments; all are strings; the second argument is written
within quotes due to space within
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Third argument: $3"
echo "Number of arguments: $#"
#\ is being used as an escape character i.e. not interpret $ as variable but
to interpret as character to print
echo "All arguments separately (\$@): $@"
echo "All arguments as a single string (\$*):$*"
```

```
#Looping through the arguments to show the diff between $* and $@
echo "Printing \$* "
for i in $*
do
        echo i is: $i
done

echo "Printing \$@ "
for i in "$@"
do
        echo i is: $i
done
```

```
# Running a command to check exit status
ls "hello.c" 2>/dev/null
echo "ls hello.c exited with  $?"

ls "non-existent-file" 2>/dev/null
echo "ls non-existent-file exited with  $?"


# Process control variables
echo "Current shell PID:  $$"
echo "My parent process id is $PPID"


# Last argument of the last command
echo "Last argument of the last command:  $_"

# Running another command to show $_ updates
echo "Hello World"
echo "Now, last argument of the last command is:  $_"
```

Functions

- Functions: allow you to group commands and reuse
  - Make scripts modular and easier to maintain
- Bash functions accept arguments just like scripts
  - $*, $#, $1, $2 … (no explicit arguments)
  - Do not return values in usual way
    - Use echo to return output.
    - Use return for exit codes

```bash
#!/bin/bash

# First we will define functions and later we will call them

# Function expecting one argument ($1 refers to it)
greet() {
   echo "Hello, $1!"
}

# Function returning Value via echo
current_date() {
   echo "Today's date is $(date +"%Y-%m-%d")"
}

# Function using return (Exit Status); also expecting one argument i.e. $1
check_file() {
   if [[ -f "$1" ]]; then
       echo "File '$1' exists."
       return 0
   else
       echo "File '$1' does not exist."
       return 1
   fi
}
```

```bash
# Function with Loop;  also expecting one argument i.e. $1
countdown() {
   local i=$1
   while [[ $i -ge 0 ]]; do
       echo "Countdown: $i"
       ((i--))
   done
}

# Recursive Function (Factorial) also possible; expecting one argument i.e. $1
factorial() {
   if [[ $1 -le 1 ]]; then
       echo 1
   else
       local temp=$(( $1 - 1 ))
       local result=$(factorial "$temp")
       echo $(( $1 * result ))
   fi
}
```

```bash
# Now we will call above functions

greet "Alice"  # Function with arguments

today=$(current_date)  # Capture function output
echo "$today"

check_file "/etc/passwd"  # Checking file existence
if [[ $? -eq 0 ]]; then
   echo "File check passed!"
else
   echo "File check failed!"
fi

countdown 3  # Loop in function

factorial_result=$(factorial 5)  # Recursion example
echo "Factorial of 5 is: $factorial_result"
```

```
#Another example of functions with arguments
function bar {
 [[ $# -ne 0 ]] || {
    echo "*** bar: must have at least 1 arg."
    return 1
 }
 echo "$@" #prints the arguments
 # no explicit return; "return 0" is implicit
}
echo "calling bar with no arguments, it should fail"
if bar; then
 echo success: $?
else
 echo failure: $?
fi
#introducing an extra line via echo
echo

echo "calling bar with 3 arguments, it should succeed"
echo 'calling bar: bar  arg1 arg2 arg3'
if bar arg1 arg2 arg3; then
 echo success: $?
else
 echo failure: $?
fi
```

Local vs Gobal Variables

- Bash variables can be local, global, or exported
  - Understanding their scope and behavior is crucial for writing modular scripts
- Global Variables: Defined without the local keyword
  - Available throughout the script, including inside functions
  - Can be modified by any part of the script
  - Note: Environment variables are global
    - Inherited by any child shells or processes

- Local Variables: Defined inside a function using the local keyword
  - Scope is limited to the function where it's declared
  - Does not affect global variables of the same name
- Exported Variables: Declared using export to make them available to child processes (subshells)
  - Useful when running another script or program from the current script

```bash
#!/bin/bash

# Global variable
message="Hello from global scope"

# Function using local variable
modify_message() {
   local message="Hello from inside function"
   echo "Inside function: $message"
}

echo "Before function call: $message"
modify_message
echo "After function call: $message"  # Global variable remains unchanged


#exporting a variable X, but not Y
#introducing an extra line via echo
echo
export X="hello"
Y=5
echo "X is $X, Y is $Y"
./var-demo-child.sh
```

File Reading

- There are several ways to read a file in a Bash script
  - Command substitution and read
  - while read -r line (Safest and Recommended)

```
#!/bin/bash

########## Method-1 via command substitution#########

echo -e "\n##Method-1## \n"
res=`cat file.txt`
echo "$res"

########## Method-2 via command substitution #########
echo -e "\n##Method-2## \n"
val=$(<file.txt)
echo "$val"
```

```
########## Method-3 read from specified file,  #########
#read: This is a built-in command that reads a line from standard input
#In this case, from the file specified by redirection at the end
#line is the name of the variable where each line of input will be stored.
#You can choose any valid variable name here instead of line
echo -e "\n##Method-3## \n"
file='file.txt'
i=1
if [ -f "$file" ]; then
   while read line; do
        #Reading each line
        echo "Line No. $i : $line"
        i=$((i+1))
   done < $file
else
   echo "File not found: $file"
fi
```

```
########## Method-4 similar to above, except filename is coming as input argument.
#Notice also -r option.
#The -r option prevent backslashes in the input from being interpreted as escape characters
#Also, here lots of checks have been added, this is how you should code

echo -e "\n##Method-4## \n"
# Check if the file name is provided as an argument
if [ $# -ne 1 ]; then
   echo "Usage: $0 <filename>"
   exit 1
fi

file="$1"  # Assign the first argument to the variable 'file'

# Check if file exists
if [ ! -f "$file" ]; then
   echo "File not found: $file"
   exit 1
fi

# Read line by line from the file
#Notice the quotes around $file, this will help if there are spaces in the file name
while read -r line; do
   echo "Line: $line"
done < "$file"
```

File Writing

- Multiple ways to write to a file
- Most flexible and commonly used methods is “here” document (EOF redirection)
  - Provides multi-line input to a command or script
  - Allows writing block text without needing echo commands for each line
  - Basic Syntax

```
command <<EOF
Multi-line text
More lines here
EOF
```

    Command receives everything between <<EOF and EOF as input

    EOF is just a delimiter—can replace it with any unique string (END, DATA, etc.), but it must match at the start and end

- Can also writing to a file using echo for shorter content
- Can use printf for better formatting

```
#!/bin/bash

# Define the output file; using capitals to avoid confusion with "file" which is a command in unix
FILE="output.txt"

# 1. Overwrite the file using cat <<EOF
cat <<EOF > "$FILE"
This is the first line.
This is the second line.
EOF

echo "==> Written using cat <<EOF"

# 2. Append more lines using cat <<EOF
cat <<EOF >> "$FILE"
This content is appended.
Appending another line.
EOF

echo "==> Appended using cat <<EOF"
```

```
# 3. Append using echo (you can also overwrite with >)
echo "This is an appended line using echo." >> "$FILE"
echo "Another appended line using echo." >> "$FILE"

echo "==> Appended using echo"

# 4. Append using printf
name="Sanjit"
score=92.3578
# Using echo (not formatted properly) to show the difference
echo "echo Format Column1" "Column2" "Column3" >> "$FILE"
echo "echo Format Name: $name, Score: $score" >> "$FILE"

echo "==> Formatting via echo"

# Using printf (formatted properly)
#-10s: - means left-aligned, 10 is 10 characters, s is string output
printf "printf Name: %s, Score: %.2f\n" "$name" "$score" >> "$FILE"
printf "printf %-10s %-10s %-10s\n" "Column1" "Column2" "Column3" >> "$FILE"

echo "==> Formatting via printf"
```

```
# 5. Writing in a loop ( appends)
echo "Writing in a loop:" >> "$FILE"
while read -r line; do
   echo "$line" >> "$FILE"
done <<EOF
Loop Line 1
Loop Line 2
Loop Line 3
EOF

echo "==> Written using while loop"
```

Arrays

- Bash supports indexed arrays and associative arrays
- Indexed Arrays (Numerically Indexed)
  - Indexed by numbers starting from 0
  - Declared using parentheses: e.g. fruits=("Apple" "Banana" "Cherry")
  - Access elements using ${array[index]}
  - Get all elements: ${array[@]}
  - Get length: ${#array[@]}
  - Add/modify elements: fruits[3]="Orange"
  - Remove an element: unset fruits[1]

- Associative Arrays (Key-Value Pairs)
  - Requires declare -A (Bash 4+)
    - Before Bash 4: Only indexed arrays were available
  - E.g.

    ```
    declare -A capitals
    capitals["France"]="Paris"
    capitals["Japan"]="Tokyo"
    ```
  - Access values: ${array["key"]}
  - Get all keys: ${!array[@]}
  - Get all values: ${array[@]}
  - Remove a key: unset array["key"]

```bash
#!/usr/bin/env bash
# --- Indexed Array ---
echo "### Indexed Array Example ###"

# Declare an indexed array
fruits=("Apple" "Banana" "Cherry")

# Add an element at index 3
fruits[3]="Orange"

# Print all elements
echo "All Fruits: ${fruits[@]}"

# Print the length of the array
echo "Number of Fruits: ${#fruits[@]}"

# Loop through the indexed array
echo "List of Fruits:"
for fruit in "${fruits[@]}"; do
   echo "$fruit"
done

# Remove an element
unset fruits[1]
echo "After removing Banana: ${fruits[@]}"
echo ""
```

```bash
# --- Associative Array ---
echo "### Associative Array Example ###"

# Declare an associative array (Requires declare -A)
declare -A capitals

# Assign values
capitals["France"]="Paris"
capitals["Japan"]="Tokyo"
capitals["USA"]="Washington D.C."
capitals["India"]="New Delhi"

# Print all keys
echo "Countries: ${!capitals[@]}"

# Print all values
echo "Capitals: ${capitals[@]}"

# Access a single value
echo "Capital of India: ${capitals["India"]}"

# Loop through the associative array
echo "Country - Capital List:"
for country in "${!capitals[@]}"; do
   echo "$country - ${capitals[$country]}"
done

# Remove an element
unset capitals["USA"]
echo "After removing USA: ${!capitals[@]}"
```

References

1. https://linuxconfig.org/bash-scripting-tutorial (a good beginner's guide)
2. https://www.javatpoint.com/bash (another guide)
3. https://tldp.org/LDP/abs/html/ (more advanced)