CHAPTER 4 : BASIC SHELL PROGRAMMING

By Hansika Bhambhaney

OBJECTIVES:

1.Write and run basic shell scripts.

2.Use and define shell functions.

3.Handle variables and script inputs.

4.Use $@ and $* correctly.

5.Perform string operations and pattern matching.

6.Use command substitution.

7.Navigate directories with pushd and popd.

4.1 SHELL SCRIPT

A shell script is a program written for the shell, a command-line interpreter that provides a user interface for Unix/Linux. Shell scripts are typically used to:

- **Automate repetitive tasks**
  Tasks that are performed frequently can be written once in a script and reused multiple times, improving efficiency.

- **Perform system administration (backups, monitoring)**
  Shell scripts are essential for maintaining and monitoring systems, handling log files, checking system health, and performing backups.

- **Combine and execute multiple shell commands**
  Rather than executing each command one at a time, a shell script allows users to bundle them and run as a single process.

- **Handle conditional logic and looping over files or inputs**
  Shell scripts support decision-making (if-else), loops (for, while), and input handling, making them powerful tools for dynamic task execution.

Structure of a Shell Script:

bash

```bash
#!/bin/bash        # Shebang – defines the interpreter
# Comment line
command_1
command_2
...
```

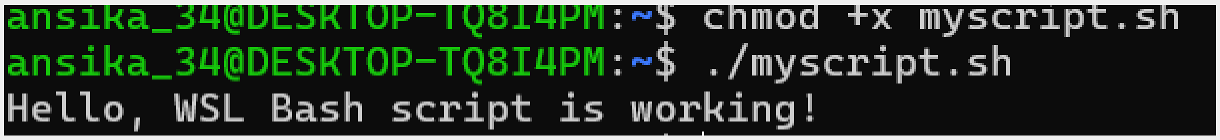
Shebang (#!)

- First line of a script, tells the OS which interpreter to use.
- Example: `#!/bin/bash` for Bash shell.

Example: Hello Script

```
#!/bin/bash

# A simple script

echo "Hello hansika , WSL Bash script is working!"
```

OUTPUT :



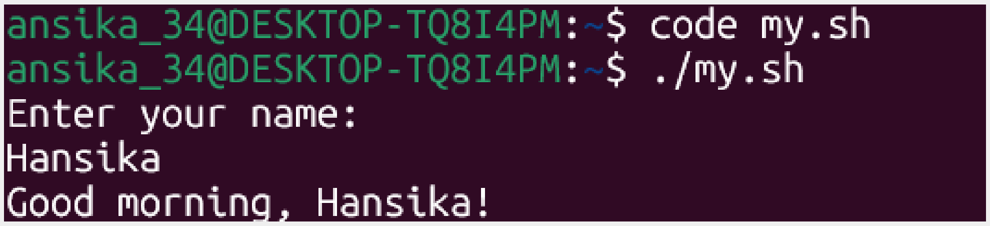
```
ansika_34@DESKTOP-TQ8I4PM:~$ chmod +x myscript.sh
ansika_34@DESKTOP-TQ8I4PM:~$ ./myscript.sh
Hello, WSL Bash script is working!
```

Example: Script with Variables and Input

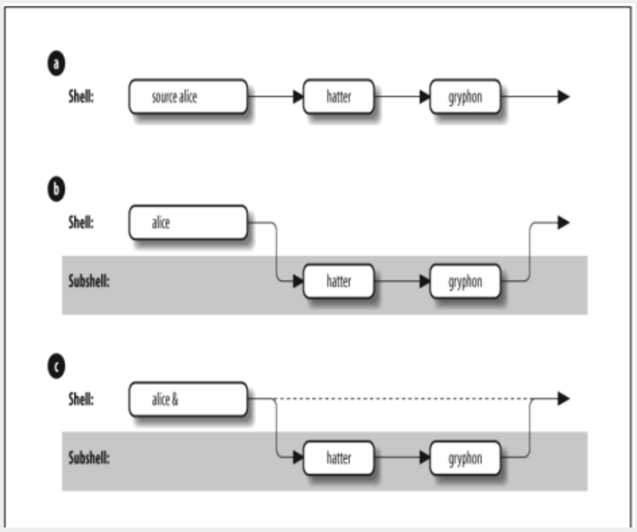
```
#!/bin/bash
```

```
echo "Enter your name:"

read name

echo "Good morning, $name!"
```

OUTPUT:



```
ansika_34@DESKTOP-TQ8I4PM:~$ code my.sh
ansika_34@DESKTOP-TQ8I4PM:~$ ./my.sh
Enter your name:
Hansika
Good morning, Hansika!
```



4.1.1 Functions in Shell Scripts

Shell functions allow code reuse, modularity, and better readability.

Functions are essential because they allow you to:

- Organize code better
- Avoid repeating code (DRY principle)
- Improve readability and debugging
- Handle modular tasks like parsing FASTA/FASTQ files, filtering sequences, formatting output, etc.

SYNTAX :

```bash
function function_name {
    # code block
}
```

Example 1:Count DNA Sequences in a FASTA File

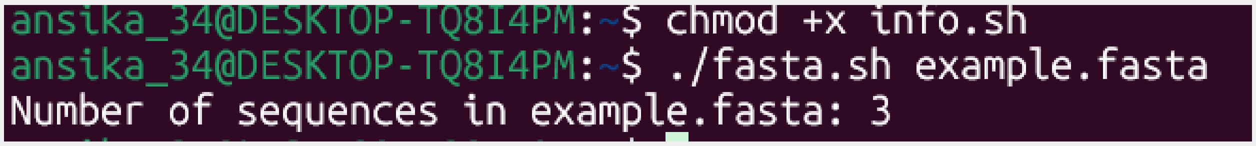
```bash
#!/bin/bash

count_sequences() {
    file=$1
    count=$(grep -c "^>" "$file")
    echo "Number of sequences in $file: $count"
}

# Usage:
count_sequences "example.fasta"
```

- `grep -c "^>"` counts lines starting with `>` (each represents a sequence header in FASTA).
- This is useful for validating inputs in sequence analysis pipelines.

EXAMPLE 2:Get GC Content of a DNA Sequence File

```bash
#!/bin/bash

calculate_gc_content() {
    file=$1
    seq=$(grep -v "^>" "$file" | tr -d '\n')
    gc=$(echo "$seq" | grep -o "[GCgc]" | wc -l)
    total=$(echo "$seq" | wc -c)
    gc_content=$(echo "scale=2; $gc*100/$total" | bc)
    echo "GC Content: $gc_content%"
}

# Usage:
calculate_gc_content "example.fasta"
```

OUTPUT:



```
ansika_34@DESKTOP-TQ8I4PM:~$ chmod +x info.sh
ansika_34@DESKTOP-TQ8I4PM:~$ ./fasta.sh example.fasta
Number of sequences in example.fasta: 3
```

4.2 SHELL VARIABLES

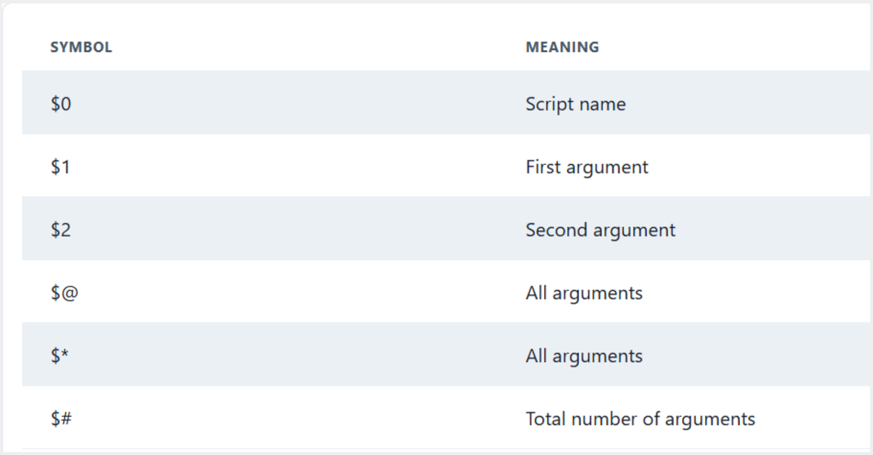
Shell variables are fundamental to any shell script or interactive Bash session. They are used to store, retrieve, and manipulate data such as strings, numbers, file names, paths, and user input.

```bash
name="Hansika"
echo "Hello, $name"
```

- No spaces around =
- Variables are accessed using $ before their name.

4.2.1 Positional Parameters

When a shell script is executed, the arguments passed to it are automatically stored in positional.



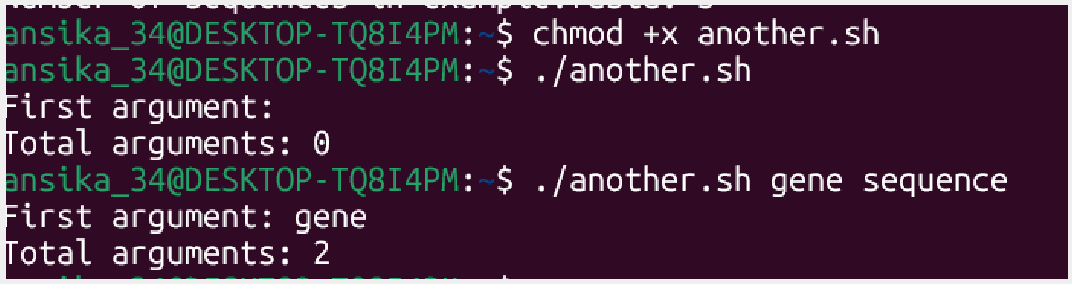
SYMBOL	MEANING
$0	Script name
$1	First argument
$2	Second argument
$@	All arguments
$*	All arguments
$#	Total number of arguments

EXAMPLE:

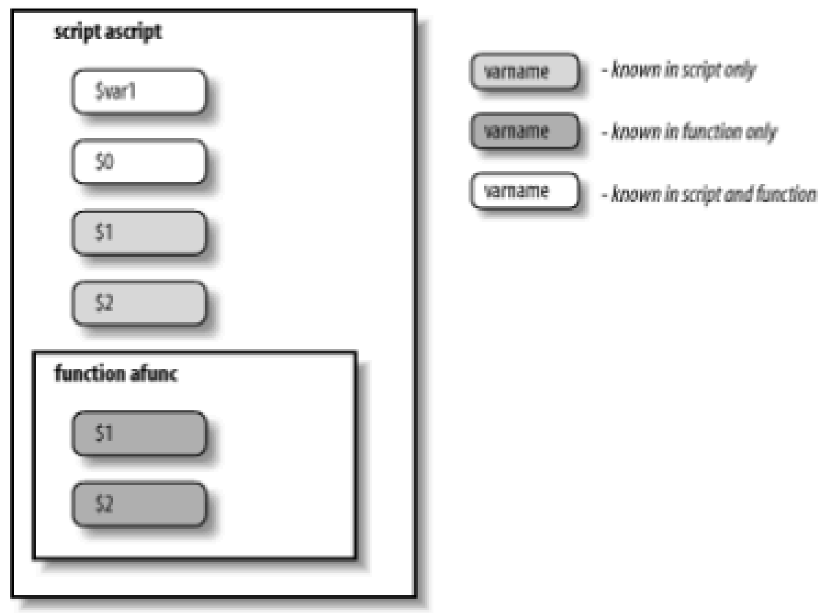
```
#!/bin/bash
```

```
echo "First argument: $1"

echo "Total arguments: $#"
```

OUTPUT :



```
ansika_34@DESKTOP-TQ8I4PM:~$ chmod +x another.sh
ansika_34@DESKTOP-TQ8I4PM:~$ ./another.sh
First argument:
Total arguments: 0
ansika_34@DESKTOP-TQ8I4PM:~$ ./another.sh gene sequence
First argument: gene
Total arguments: 2
```



script ascript
$var1
$0
$1
$2
function afunc
$1
$2
varname - known in script only
varname - known in function only
varname - known in script and function

Figure 4-2. Functions have their own positional parameters

4.2.2 LOCAL VARIABLES IN FUNCTIONS

A local statement inside a function definition makes the variables involved all become local to that function. The ability to define variables that are local to "subprogram" units (procedures, functions, subroutines, etc.) is necessary for writing large programs, because it helps keep subprograms independent of the main program and of each other.

```bash
my_function() {
  local var="Inside Function"
  echo $var
}
```

This avoids conflicts with other parts of the script.

Key Characteristics:

- Declared using the `local` keyword
- Scope is limited to the function where they are defined
- Prevent unintended changes to variables with the same name in other parts of the script

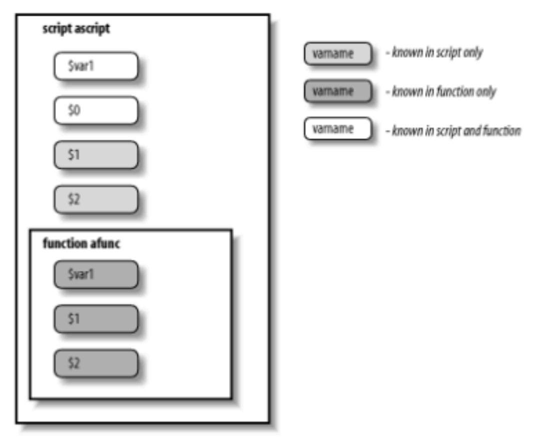
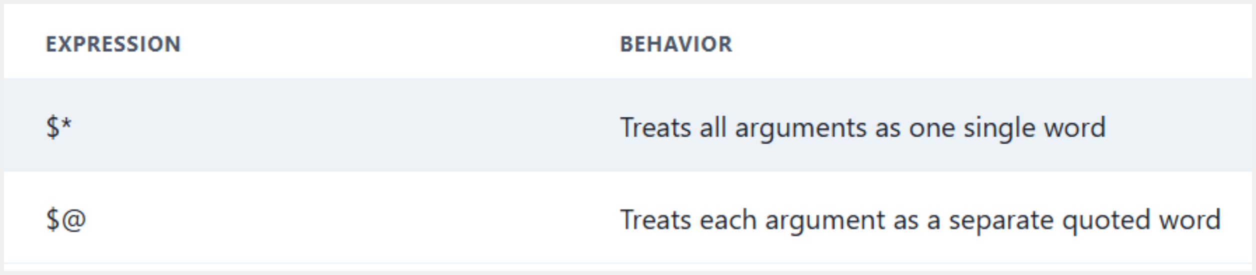

script ascript
$var1
$0
$1
$2
function afunc
$var1
$1
$2
varname - known in script only
varname - known in function only
varname - known in script and function

Figure 4-3. Functions can have local variables

4.2.3 Quoting with `$@` and `$*`

Both $@ and $* represent all positional arguments, but:
IFS=, echo "$*"



EXPRESSION	BEHAVIOR
$*	Treats all arguments as one single word
$@	Treats each argument as a separate quoted word

This matters when arguments contain spaces.

4.2.4 More on Variable Syntax

Bash provides **advanced forms of variable usage**, such as:

- **Default values:**
  `${var:-default}` → Use default if `var` is unset
- **Substring operations:**
  `${var:0:4}` → Extract substring from position 0 of length 4
- **Length of a variable:**
  `${#var}` → Returns the length of the value

4.3 STRING OPERATORS

The curly-bracket syntax allows for the shell's string operators. String operators allow you to manipulate values of variables in various useful ways without having to write full-blown programs or resort to external UNIX utilities.

In particular, string operators let us do the following:

• Ensure that variables exist (i.e., are defined and have non-null values)

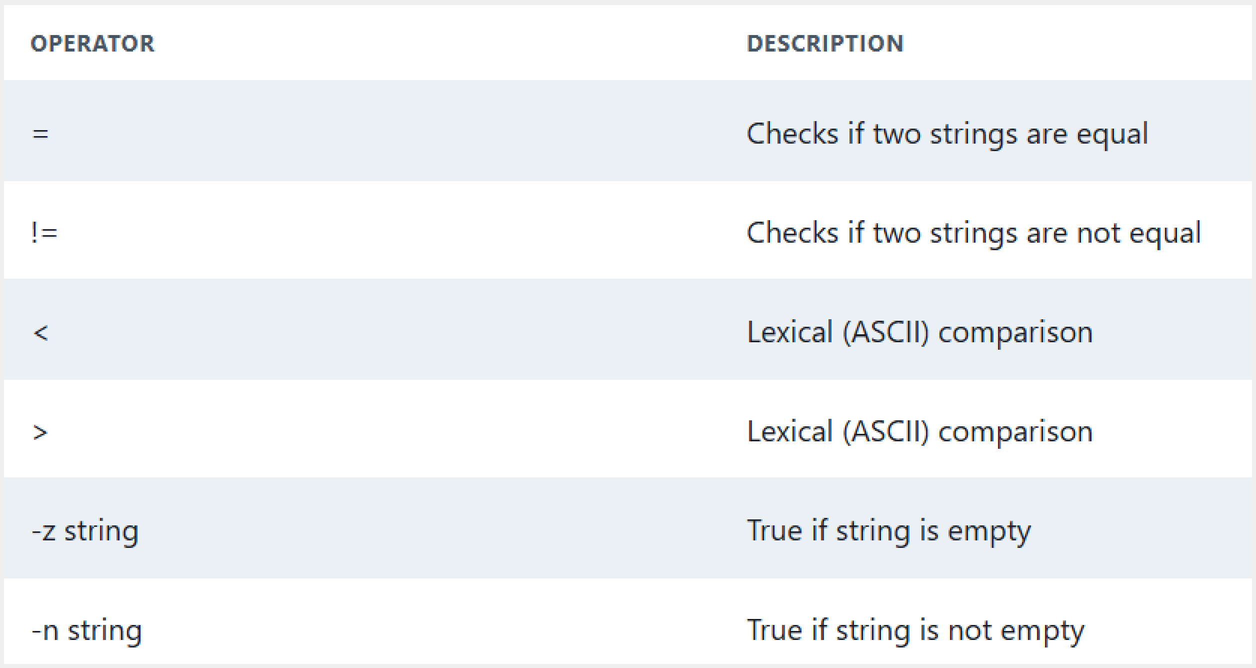
• Set default values for variables

• Catch errors that result from variables not being set • Remove portions of variables' values that match patterns.

4.3.1 Syntax of String Operators

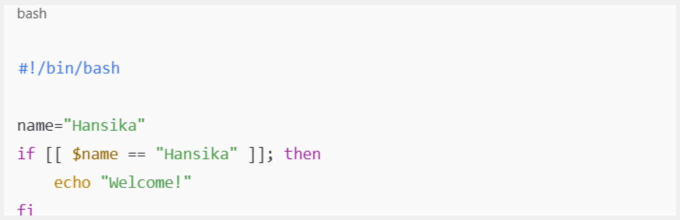
Bash uses both `[[ ... ]]` conditional expressions and parameter expansion (`${...}`) for string operations.

Examples of string operations:



OPERATOR	DESCRIPTION
=	Checks if two strings are equal
!=	Checks if two strings are not equal
<	Lexical (ASCII) comparison
>	Lexical (ASCII) comparison
-z string	True if string is empty
-n string	True if string is not empty

Example: String Comparison



bash

```bash
#!/bin/bash

name="Hansika"
if [[ $name == "Hansika" ]]; then
    echo "Welcome!"
fi
```

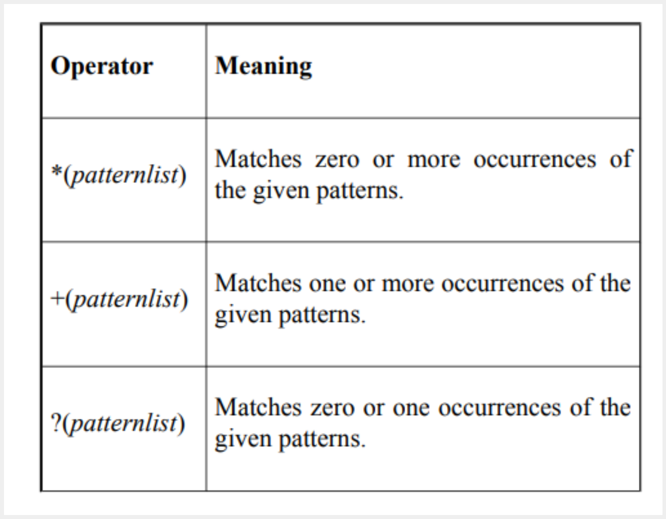
4.3.2 Patterns and Pattern Matching

Pattern matching is a technique that allows Bash to compare strings not by exact match, but by checking if a string fits a certain pattern or wildcard expression.

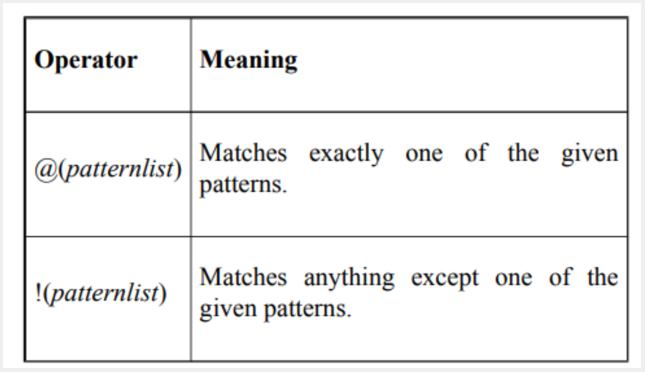
It is commonly used to:

- Match file names (like `*.fasta`)
- Detect string prefixes/suffixes (e.g., `ATG*`)
- Filter or extract specific text data
- Validate sequences, headers, gene IDs, etc.

Pattern matching does not use regular expressions by default; instead, Bash uses glob patterns (like `*`, `?`, `[ ]`) for matching.



Operator	**Meaning**
*(*patternlist*)	Matches zero or more occurrences of the given patterns.
+(*patternlist*)	Matches one or more occurrences of the given patterns.
?(*patternlist*)	Matches zero or one occurrences of the given patterns.



Operator	**Meaning**
@(*patternlist*)	Matches exactly one of the given patterns.
!(*patternlist*)	Matches anything except one of the given patterns.

PATTERN	MEANING
*	Matches zero or more characters
?	Matches exactly one character
[abc]	Matches any one character in set
[a-z]	Matches a range of characters
[^a-z]	Matches any character except in range

Some examples of these include:
• *(alice|hatter|hare) would match zero or more occurrences of alice, hatter, and hare. So it would match the null string, alice, alicehatter, etc.
• +(alice|hatter|hare) would do the same except not match the null string.
• ?(alice|hatter|hare) would only match the null string, alice, hatter, or hare.
• @(alice|hatter|hare) would only match alice, hatter, or hare.
• !(alice|hatter|hare) matches everything except alice, hatter, and hare.

Real-World Bioinformatics Relevance

In **bioinformatics**, pattern matching is especially useful when:

- Filtering FASTA/FASTQ headers (e.g., `>chr*`)
- Matching gene or protein IDs (e.g., `BRCA*`)
- Validating DNA/RNA sequence start codons (e.g., `ATG*`)
- Finding file types in bulk processing (e.g., `*.fasta`, `*.vcf`)

```
sequence="ATGCGTACG"
```

```
if [[ $sequence == ATG* ]]; then
    echo "Starts with ATG (start codon)"
fi
```

4.3.3 Length Operator

Get the **length of a string** using `${#var}`:

```bash
seq="AGCTAGC"
echo "Length: ${#seq}"  # Output: 7
```

4.3.4 Extended Pattern Matching in Bash

Extended pattern matching in Bash allows you to perform advanced string pattern comparisons using logical expressions and repetition rules — similar to regex but simpler and native to Bash.

It enhances the basic wildcard system (`*`, `?`, `[abc]`) and is useful in file filtering, data validation, and string logic inside scripts.

```
shopt -s extglob
files=( "a.jpg" "b.gif" "c.png" "d.pdf" "ee.pdf" )

for file in "${files[@]}"; do
  if [[ "$file" == *(.jpg|.gif) ]]; then
    echo "Match: $file"
  fi
done
```

This code snippet first enables extended globbing. Then, it iterates through an array of filenames. The `if` statement uses the `?(pattern)` construct, which in this case matches files ending with either `.jpg` or `.gif`. If a match is found, it prints the filename. The `.` in `*(.jpg|.gif)` is not a part of the globbing pattern, it is to match a literal dot.

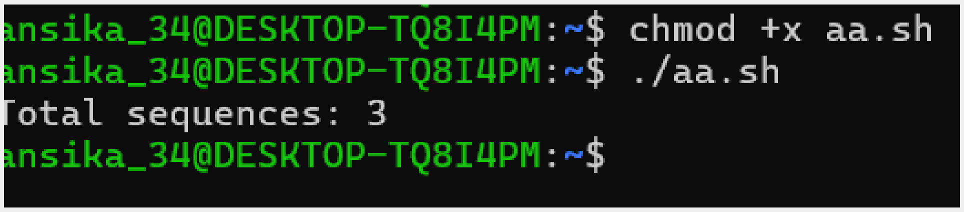
4.4 COMMAND SUBSTITUTION

- Command substitution allows capturing the output of a command and using it as a value in a script.
- It enables dynamic assignment of values based on system state, file content, or processing result.
- It is essential for automation, script flexibility, and dynamic data handling.
- The preferred syntax for command substitution is `$(command)` because it is cleaner and supports nesting.
- The older syntax is `` `command` ``, which is harder to read and maintain.
- Command substitution can be used in variable assignment, command arguments, or string construction.
- It is evaluated before the main command runs, replacing the entire expression with the command's output.
- It is widely used in bioinformatics scripts for tasks such as counting sequences or capturing filenames.
- For example:

```bash
#!/bin/bash

file="example.fasta"

```

```
if [[ ! -f "$file" ]]; then
    echo "Error: File '$file' not found!"
    exit 1
fi

count=$(grep -c "^>" "$file")
echo "Total sequences: $count"

```

OUTPUT :



```
ansika_34@DESKTOP-TQ8I4PM:~$ chmod +x aa.sh
ansika_34@DESKTOP-TQ8I4PM:~$ ./aa.sh
Total sequences: 3
ansika_34@DESKTOP-TQ8I4PM:~$
```

4.5 Directory Stack in Bash (pushd and popd)

Concept of a Stack (LIFO)

- A **stack** is a Last-In-First-Out (LIFO) structure, like a stack of plates.
- **push** adds a new item on top.
- **pop** removes the item from the top.
- In directory navigation, this lets us **remember and return to previous folders** in order.

🔷 Why Use pushd and popd in Bash?

- Bash has `cd -` to switch to the **previous directory**, but it remembers **only one**.

- For **multi-level navigation**, we need a **custom stack**.
- Using `pushd`/`popd`, we can **push directories onto a stack** and **pop back through history** easily.

🔷 Stack Setup

- Declare a global stack variable in `.bash_profile`:

```bash
DIR_STACK=""
export DIR_STACK
```

- This ensures:
  - The stack is **initialized once per login**
  - It is **available to all child shells**

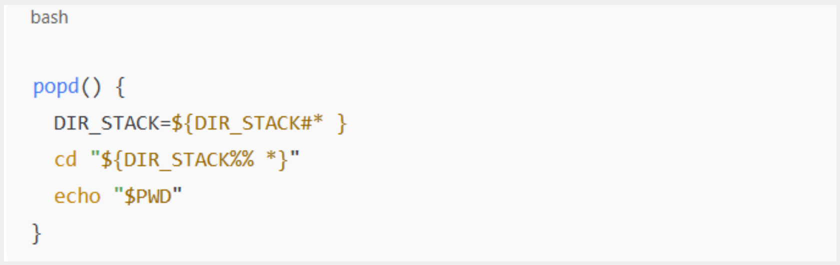
🔧 Custom pushd Implementation

```
    pushd() {
  dirname=$1
  DIR_STACK="$dirname ${DIR_STACK:-$PWD' '}"
  cd "${dirname:?"missing directory name."}"
  echo "$DIR_STACK"
}
```

Key Parts:

- `$1`: Directory passed to the function
- `${DIR_STACK:-$PWD' '}`: If `DIR_STACK` is empty, use current directory
- `cd "${dirname:?}"`: If no directory is given, throw error
- Appends to the front of the stack and changes directory
- The stack preserves navigation history, allowing multi-level return using `popd`.
- Helps navigate deeply nested directories efficiently in custom Bash setups

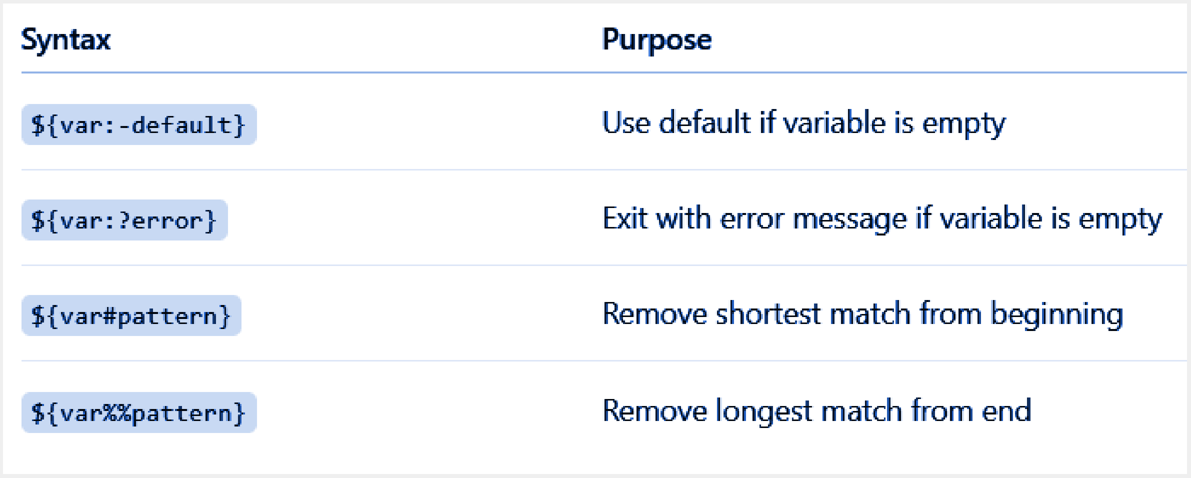
🔧 Custom popd Implementation



```bash
popd() {
  DIR_STACK=${DIR_STACK#* }
  cd "${DIR_STACK%% *}"
  echo "$PWD"
}
```

Key Parts:

- `${DIR_STACK#* }`: Removes the first item (i.e., top of stack)
- `${DIR_STACK%% *}`: Extracts the new top item
- Changes to that directory and prints it



Syntax	Purpose
`${var:-default}`	Use default if variable is empty
`${var:?error}`	Exit with error message if variable is empty
`${var#pattern}`	Remove shortest match from beginning
`${var%%pattern}`	Remove longest match from end

EXAMPLE SESSION :

```
$ pwd
/home/hansika

$ pushd projects
/home/hansika/projects /home/hansika

$ pushd /etc
```

```
/etc /home/hansika/projects /home/hansika

$ popd
/home/hansika/projects /home/hansika
$ pwd
/home/hansika/projects

$ popd
/home/hansika
$ pwd
/home/hansika

$ popd
bash: cd: too few arguments
```

EXPLANATION:

1.Each `pushd` command adds the new directory to the top of the stack and switches to it, preserving the previous directories in `DIR_STACK`.

2.Each `popd` command removes the top directory from the stack and switches back to the next one stored.

3.When all entries are popped, further `popd` calls result in an error (`cd: too few arguments`) because the stack is empty.

4.This simulates **multi-level backtracking** through directories, much more powerful than `cd -` which remembers only one level.

5.The directory stack behaves like **LIFO (Last-In-First-Out)** — the last pushed directory is the first one popped.

6.This approach is useful in scripts or manual navigation when working across **multiple nested project folders** without losing track of the path history.

SUMMARY

This chapter covers the basics of writing and running shell scripts in Bash. Shell scripting is used to automate tasks, manage files, and handle system processes. A script begins with a shebang (`#!/bin/bash`), followed by commands like `echo`, `read`, and variable assignments.

It introduces functions for modular code and examples like counting DNA sequences and calculating GC content. Shell variables, positional parameters (`$1`, `$@`, `$*`), and local variables are explained for better data handling.

String operations and pattern matching (using glob patterns like `*`, `?`, `@(pattern)`) are used for filtering and validating data, especially useful in bioinformatics.

Command substitution (`$(command)`) allows storing command output in variables. The use of pushd and popd enables stack-based directory navigation, allowing users to move back through folders efficiently.

In essence, this chapter provides a foundation for automating and managing tasks using Bash scripts.