CHAPTER 7 Input/Output and Command-Line Processing

By Hansika Bhambhaney

OBJECTIVES :

1.Use input/output redirection.

2.Handle multiline input with here-docs.

3.Work with file descriptors.

4.Display and read strings using I/O commands.

5.Understand command-line parsing.

6.Apply proper quoting in commands.

7.Use internal shell commands.

8.Evaluate and run dynamic commands.

1. Input (`<`) and Output (`>`, `>>`) Redirection

Theory:

Bash, like other Unix shells, treats input and output as streams. By default:

- Standard Input (stdin) comes from the keyboard.
- Standard Output (stdout) goes to the screen.
- Standard Error (stderr) also goes to the screen.

Redirection allows you to change this default behavior.
You can redirect input from a file and redirect output to a file.

Input Redirection (`<`):

It makes a command read input from a file instead of the keyboard.

Example:

sort < names.txt

- `sort` takes input from `names.txt` instead of waiting for typed input.

Output Redirection (`>`):

It sends the output of a command to a file, overwriting the file if it exists.

Example:

echo "Hello, World!" > greeting.txt

1.If `greeting.txt` exists, it will be replaced.

2.If it doesn't exist, it will be created.

Append Output (`>>`):

This sends output to a file without deleting existing content — it **appends** to the end.

EXAMPLE:

echo "New line" >> greeting.txt

2. Redirecting Standard Error (stderr)

Every process in Bash has **three standard file descriptors**:

- `0` → Standard Input (keyboard)
- `1` → Standard Output (screen)
- `2` → Standard Error (errors)

If you want to separate errors from output, or capture both, you must use redirection with these numbers.

Redirect stderr only:

ls file1 file2 2> errors.txt

EXPLANATION:

1.Output (list of files) goes to screen.

2.Error message (if file doesn't exist) goes to `errors.txt`.

Redirect stdout and stderr together:

ls file1 file2 > output.txt 2>&1

EXPLANATION:

`> output.txt` sends stdout to file.

`2>&1` sends stderr to where stdout is already going (i.e., `output.txt`).

3. Here-Documents (<<)

A here-document is a way to provide multi-line input directly inside a script or command, without needing an external file.

It uses the syntax:

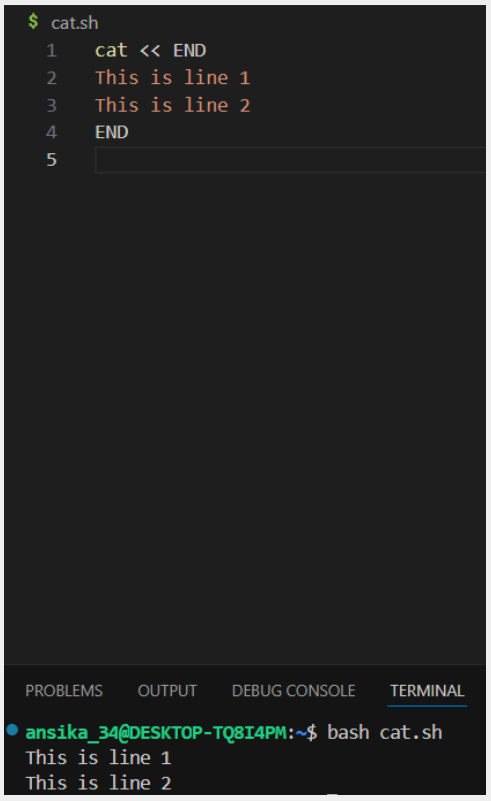
command << delimiter

multi-line text

delimiter

The shell passes the block of text to the command as standard input.

EXAMPLE:



```
cat << END
This is line 1
This is line 2
END
```

```
ansika_34@DESKTOP-TQ8I4PM:~$ bash cat.sh
This is line 1
This is line 2
```

`cat` receives 2 lines of input as if they were typed from a file.

`END` is a marker and must match exactly on the closing line.

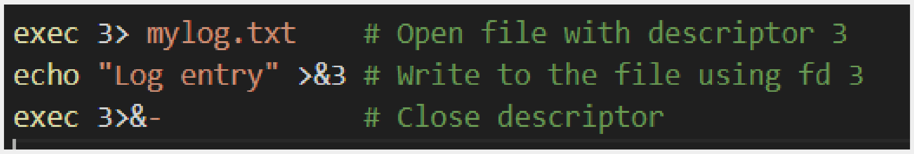
4. File Descriptors

In Bash, you can create and control custom file descriptors (besides the default 0, 1, 2). This is useful for:

- Advanced script control
- Opening multiple input/output streams
- Managing temporary file I/O

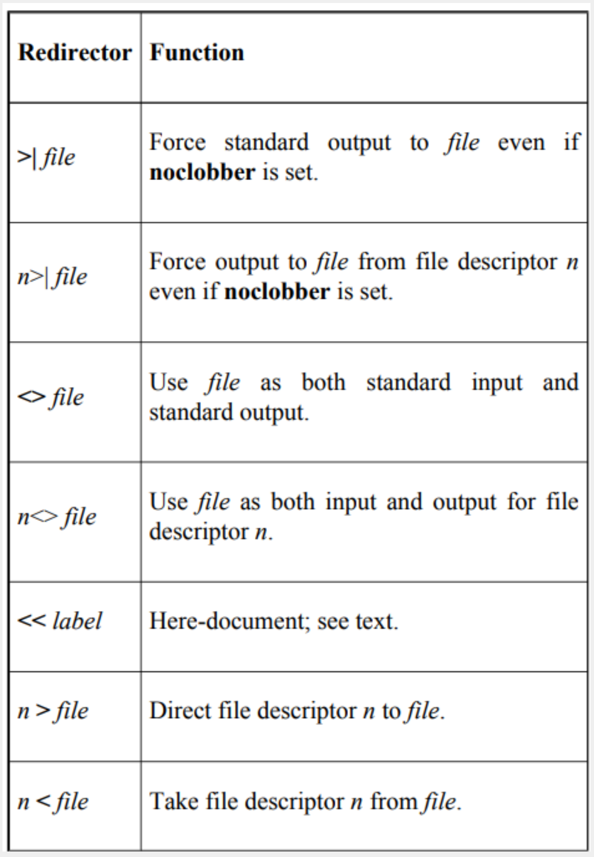
Descriptors are assigned using `exec`.

Writing with a new file descriptor:

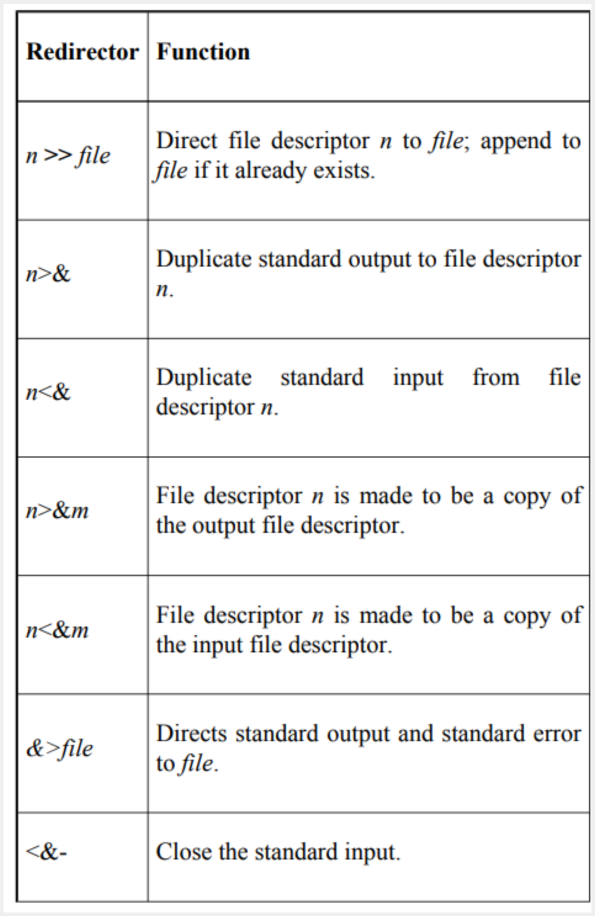


```
exec 3> mylog.txt    # Open file with descriptor 3
echo "Log entry" >&3 # Write to the file using fd 3
exec 3>&-            # Close descriptor
```

I/O REDIRECTORS TABLE



Redirector	**Function**
>| *file*	Force standard output to *file* even if **noclobber** is set.
n>| *file*	Force output to *file* from file descriptor *n* even if **noclobber** is set.
<> *file*	Use *file* as both standard input and standard output.
n<> *file*	Use *file* as both input and output for file descriptor *n*.
<< *label*	Here-document; see text.
n > *file*	Direct file descriptor *n* to *file*.
n < *file*	Take file descriptor *n* from *file*.



Redirector	**Function**
n >> *file*	Direct file descriptor *n* to *file*; append to *file* if it already exists.
n>&	Duplicate standard output to file descriptor *n*.
n<&	Duplicate standard input from file descriptor *n*.
n>&*m*	File descriptor *n* is made to be a copy of the output file descriptor.
n<&*m*	File descriptor *n* is made to be a copy of the input file descriptor.
&>*file*	Directs standard output and standard error to *file*.
<&-	Close the standard input.

7.4 String I/O

String I/O refers to handling text-based input and output in the shell. Bash provides three primary tools for this:

1. `echo` → to print simple strings
2. `printf` → to print formatted output
3. `read` → to take user input

These are especially useful in scripts and interactive terminals.

1. echo – Displaying Text

Theory:

- `echo` prints a line of text to the standard output.
- It's the simplest form of output.
- It's often used to display messages, status updates, and debugging info in scripts.

Syntax:

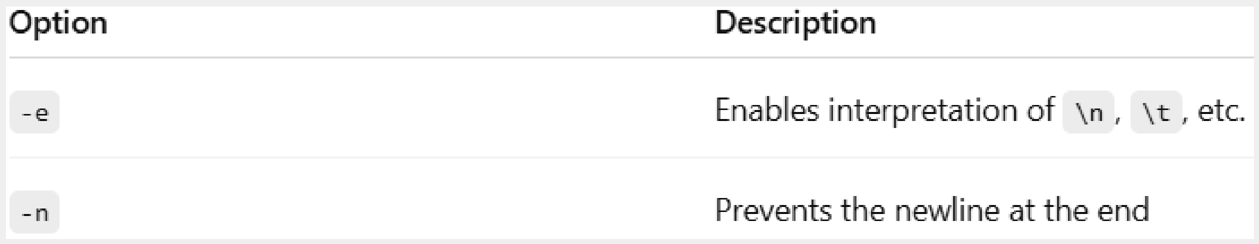
```
echo [options] [string]
```

Examples:

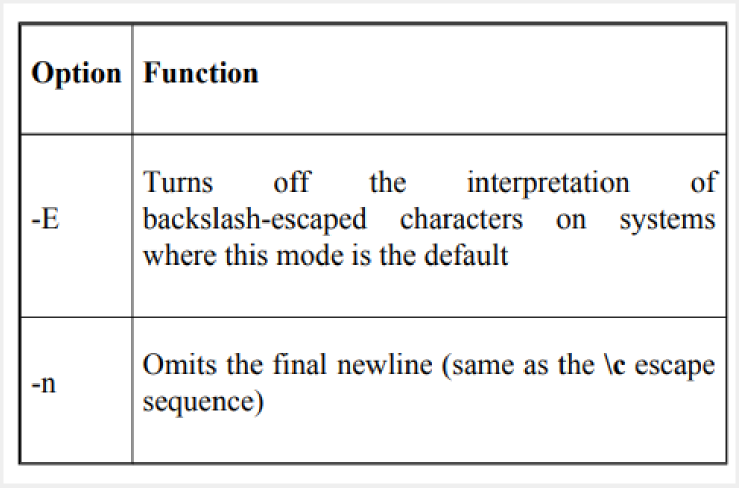
1.echo "Hello, World!"

Output: Hello, World!

2.echo -e "This is line 1\nThis is line 2"



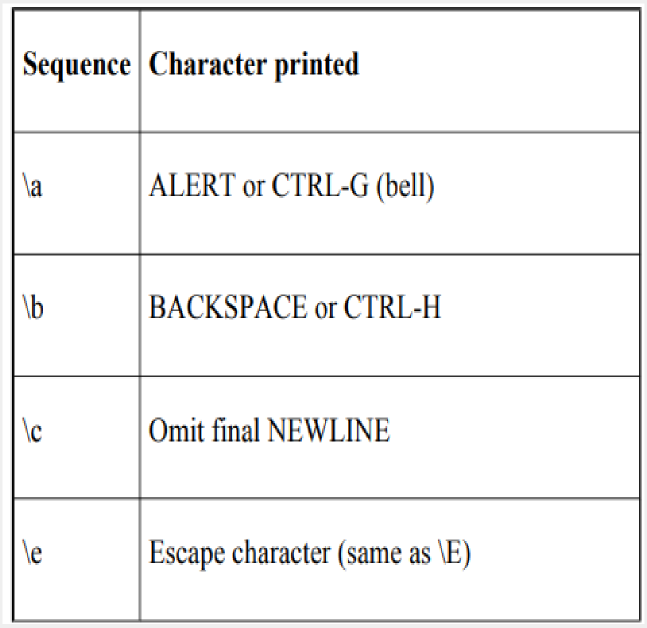
Option	Description
`-e`	Enables interpretation of `\n`, `\t`, etc.
`-n`	Prevents the newline at the end



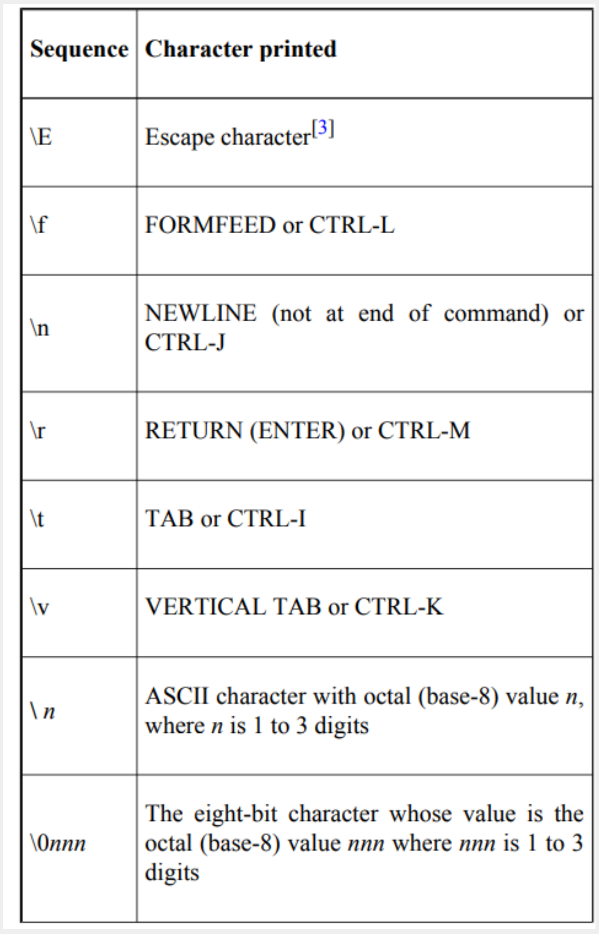
Option	Function
-E	Turns off the interpretation of backslash-escaped characters on systems where this mode is the default
-n	Omits the final newline (same as the **\c** escape sequence)

7.4.1 echo escape sequences

echo accepts a number of escape sequences that start with a backslash. These sequences exhibit fairly predictable behavior, except for \f: on some displays, it causes a screen clear, while on others it causes a line feed. It ejects the page on most printers. \v is somewhat obsolete; it usually causes a line feed.



Sequence	Character printed
\a	ALERT or CTRL-G (bell)
\b	BACKSPACE or CTRL-H
\c	Omit final NEWLINE
\e	Escape character (same as \E)



Sequence	Character printed
\E	Escape character[3]
\f	FORMFEED or CTRL-L
\n	NEWLINE (not at end of command) or CTRL-J
\r	RETURN (ENTER) or CTRL-M
\t	TAB or CTRL-I
\v	VERTICAL TAB or CTRL-K
\ *n*	ASCII character with octal (base-8) value *n*, where *n* is 1 to 3 digits
\0*nnn*	The eight-bit character whose value is the octal (base-8) value *nnn* where *nnn* is 1 to 3 digits

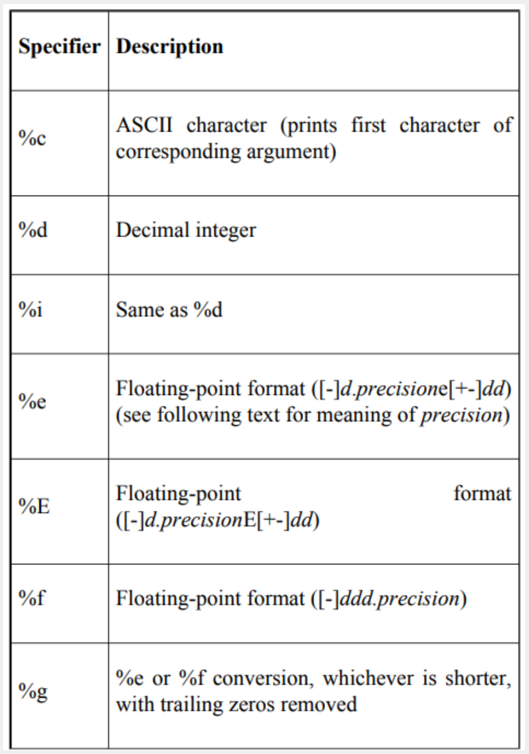
2. `printf` – Formatted Output

Theory:

- `printf` (like in C) gives you precise control over formatting.
- It's better than `echo` when printing numbers, aligned columns, or variables with specific formatting.

Syntax:

printf "format-string" [arguments...]



Specifier	Description
%c	ASCII character (prints first character of corresponding argument)
%d	Decimal integer
%i	Same as %d
%e	Floating-point format ([-]*d.precision*e[+-]*dd*) (see following text for meaning of *precision*)
%E	Floating-point format ([-]*d.precision*E[+-]*dd*)
%f	Floating-point format ([-]*ddd.precision*)
%g	%e or %f conversion, whichever is shorter, with trailing zeros removed

EXAMPLE:

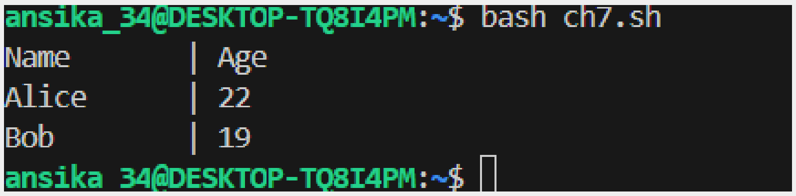
printf "Hello, %s!\n" "Hansika"

Output: Hello, Hansika!

EXAMPLE 2:

```
printf "%-10s | %-5s\n" "Name" "Age"
printf "%-10s | %-5d\n" "Alice" 22
printf "%-10s | %-5d\n" "Bob" 19
```

OUTPUT:



```
ansika_34@DESKTOP-TQ8I4PM:~$ bash ch7.sh
Name        | Age
Alice       | 22
Bob         | 19
ansika_34@DESKTOP-TQ8I4PM:~$
```

3. `read` – Taking Input from User

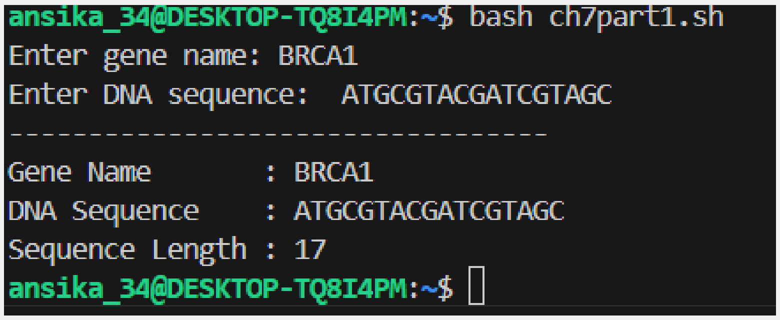
- `read` is used to get input from the keyboard (stdin).
- It can read a line into one or more variables.

Syntax:

read [options] variable_name

```
#!/bin/bash

# Prompt for gene name
read -p "Enter gene name: " gene

# Prompt for DNA sequence
read -p "Enter DNA sequence: " sequence

# Calculate the length of the DNA sequence
length=${#sequence}

# Display the information
echo "----------------------------------"
echo "Gene Name       : $gene"
echo "DNA Sequence    : $sequence"
echo "Sequence Length : $length"
```

OUTPUT:



```
ansika_34@DESKTOP-TQ8I4PM:~$ bash ch7part1.sh
Enter gene name: BRCA1
Enter DNA sequence:  ATGCGTACGATCGTAGC
----------------------------------
Gene Name       : BRCA1
DNA Sequence    : ATGCGTACGATCGTAGC
Sequence Length : 17
ansika_34@DESKTOP-TQ8I4PM:~$
```

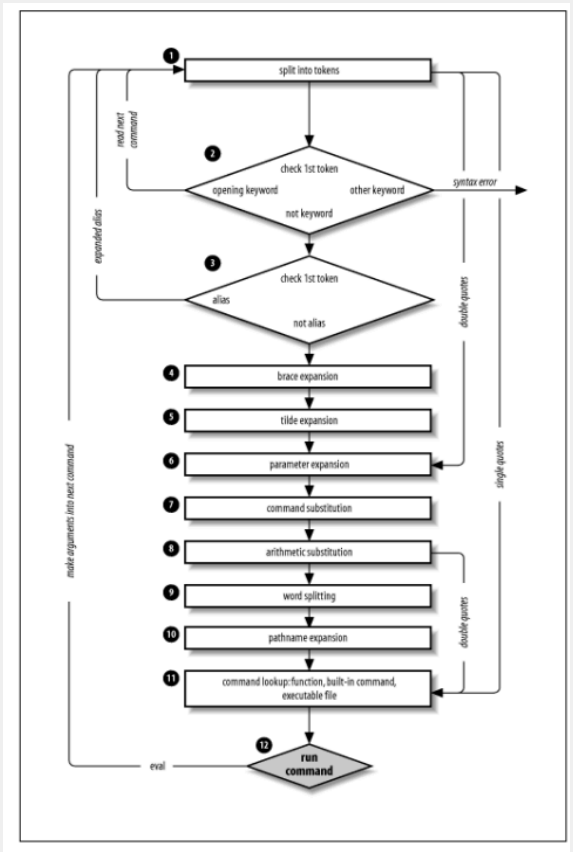
7.4 I/O redirection and multiple commands

This involves creating a separate process to do the reading. However, it is usually more efficient to do it in the same process; bash gives us four ways of doing this.

CODE :

```
findterm() {
    TERM="vt100"  # Default terminal type
    line=$(tty)   # Get current terminal device (e.g., /dev/pts/0)

    while read -r dev termtype; do
        if [ "$dev" = "$line" ]; then
            TERM="$termtype"
            echo "TERM set to $TERM."
            break
        fi
    done < /etc/terms
}
```

7.5 COMMAND LINE PROCESSING



Bash processes a command line **in several steps** before execution:

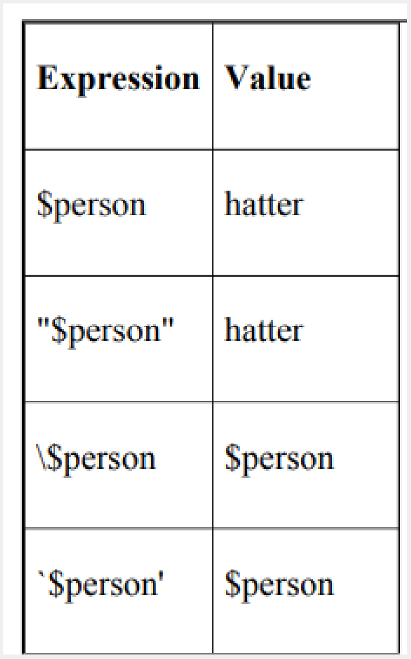
1. **Tokenizing** (splitting into words)
2. **Expanding** (variables, wildcards, command substitutions, etc.)
3. **Quoting** (controlling expansion)
4. **Command lookup**
5. **Execution**

Understanding how Bash interprets a command line helps avoid bugs, especially when scripting for tasks like file parsing, command automation, or bioinformatics pipelines.

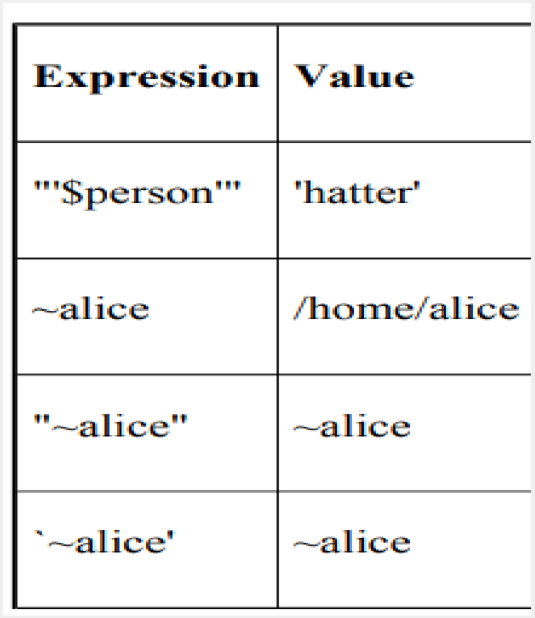
1. Quoting Mechanisms:

Quoting controls **how Bash interprets special characters**, like `$`, `*`, space, etc.

Types of Quotes:



Expression	Value
$person	hatter
"$person"	hatter
\$person	$person
`$person'	$person



Expression	Value
"'$person'"	'hatter'
~alice	/home/alice
"~alice"	~alice
`~alice'	~alice

EXAMPLE:

```
gene="BRCA1"
echo "Searching for $gene in database"
grep "$gene" sequences.fasta
```

1.Double quotes protect the variable (in case it contains spaces or special characters).
2.Single quotes would **prevent expansion**: `grep '$gene'` would search for the literal string `$gene`.

2. **`command`, `builtin`, and `enable`**

These are used to control how commands are executed, especially when aliases or functions may override the original behavior.

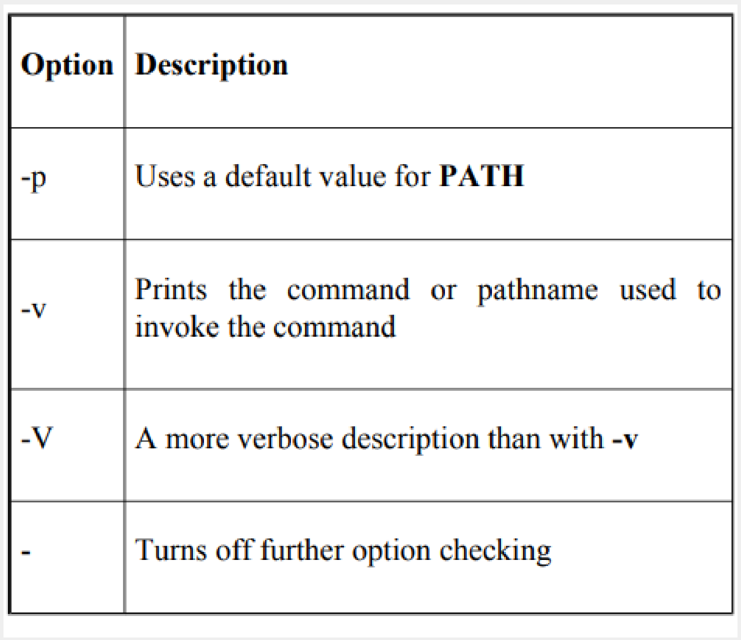
Command

Runs a command **ignoring functions or aliases** of the same name.

EXAMPLE:

```
alias grep='grep --color=always'
command grep "ATG" sequences.fasta
```

–This runs the original `grep`, ignoring the alias.



Option	**Description**
-p	Uses a default value for **PATH**
-v	Prints the command or pathname used to invoke the command
-V	A more verbose description than with **-v**
-	Turns off further option checking

`builtin`

Forces Bash to run its **built-in version** of a command, useful when a program with the same name exists in the system.

Example:

builtin echo "Gene search started"

`Enable`

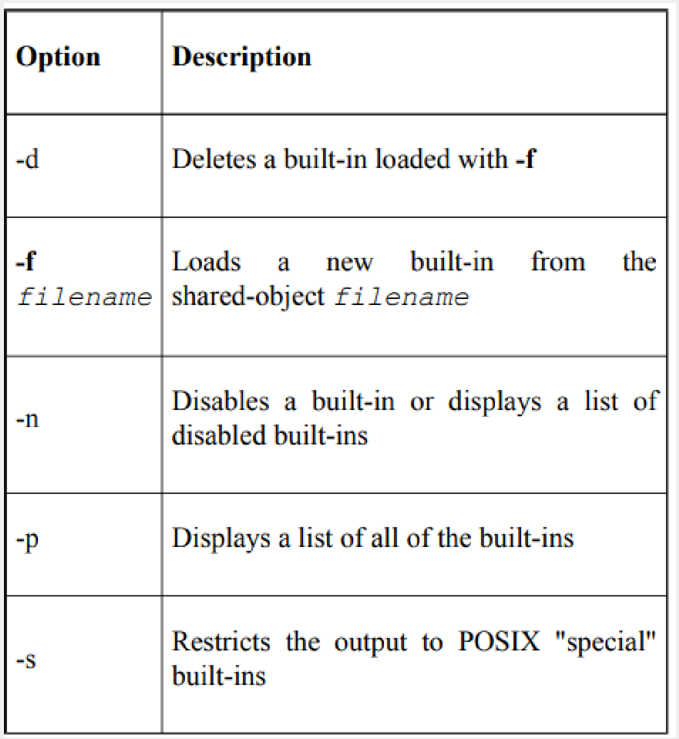
`enable` is a **built-in Bash command** used to:

- **View** all shell built-in commands.

- **Enable** or **disable** shell built-ins.
- **Replace** a built-in temporarily with an external command (like `/bin/echo`).
- **Use specific built-ins** when conflicts with external binaries or aliases occur.

SYNTAX:
enable [options] [name...]

.



Option	**Description**
-d	Deletes a built-in loaded with **-f**
-f *filename*	Loads a new built-in from the shared-object *filename*
-n	Disables a built-in or displays a list of disabled built-ins
-p	Displays a list of all of the built-ins
-s	Restricts the output to POSIX "special" built-ins

The -p option is a default path which guarantees that the command lookup will find all of the standard UNIX utilities. In this case, command will ignore the directories in your PATH.builtin is very similar to command but is more restrictive. It looks up only built-in commands, ignoring functions and commands found in PATH. The last command enables and disables shell built-ins—it is called enable. Disabling a built-in allows a shell script or executable of the same name to be run without giving a full pathname.

7.8 Evaluation of commands with eval

The `eval` command evaluates a string as a Bash command, parses it, and executes it as if it were typed directly on the command line.
SYNTAX:
eval [arguments as a string]
Bioinformatics Example: Dynamic Command Execution

```
tool="grep"
pattern="ATGC"
file="genes.fasta"

cmd="$tool '$pattern' $file"
echo "Running: $cmd"
eval $cmd
```

If `tool=grep`, this executes: `grep 'ATGC' genes.fasta`
Now suppose that we write a shell function called makecmd that reads and executes a single construct of this form. Assume that the makefile is read from standard input. The function would look like the following code.
CODE:

```
makecmd () {
    read target colon sources
    for src in $sources; do
        if [ "$src" -nt "$target" ]; then
            while read -r cmd; do
                [[ "$cmd" == $'\t'* ]] || break
                echo "$cmd"
                eval "${cmd#"$'\t'"}"
            done
            break
        fi
    done
}
```
This function reads the line with the target and sources; the variable colon is just a placeholder for the :. Then it checks each source to see if it's newer than the target, using the -nt file attribute test operator that we saw in Chapter 5. If the source is newer, it reads, prints, and executes the commands until it finds a line that doesn't start with a TAB or it reaches end-of-file. (The real make does more than this; see the exercises at the end of this chapter.) After running the commands (which are stripped of the initial TAB), it breaks out of the for loop, so that it doesn't run the commands more than once.

SUMMARY

Chapter 7 covers input/output redirection in Bash using `<`, `>`, and `>>`, handling standard error with `2>`, and combining outputs with `2>&1`. It explains here-documents (`<<`) for multiline input and using custom file descriptors with `exec`. String I/O is done using `echo`, `printf`, and `read` for output and user input. The chapter also explains how Bash processes command lines—through tokenizing, expanding, quoting, and execution. Quoting mechanisms (`'`, `"`, `` ` ``) help control expansion. The use of `command`, `builtin`, and `enable` manages command execution behavior, while `eval` is used to execute dynamically built commands, useful in automation and bioinformatics scripting.