# Chapter 6. Command-Line Options and Typed Variables

## .1Command-Line Options

In Bash scripting, command-line options are *flags* or *switches* that modify the behavior of a script. They usually start with a dash (-), such as -h, -o, -v, and so on.

They are *not* positional parameters like $1, $2, etc., which represent arguments. Instead, they are used to indicate a specific behavior or to pass optional parameters that control how the script works.

EXAMPLE:

ls -l -a

Here:

* + ls is the command
  + -l is an option for "long format"
  + -a is an option to show hidden files

### 6.1.1 OPTIONS WITH ARGUMENTS

Options with arguments are command-line flags that are followed by a value the script needs to function correctly.

WHY OPTIONS ARE USED ?

1. Let users pass input dynamically (e.g., filenames, thresholds, output paths)
2. Avoid hardcoding values in your script
3. Mimic standard Linux tools like grep, cut, bwa, etc.
4. Enable automation in pipelines and workflows (especially useful in bioinformatics!)

# Implementing Options with Arguments Using getopts

******Syntax:**

while getopts "i:o:t:" opt; do case $opt in

i) input\_file="$OPTARG" ;;

o) output\_file="$OPTARG" ;;

t) threads="$OPTARG" ;;

\?) echo "Invalid option: -$OPTARG" >&2; exit 1 ;; esac

done

**The "i:o:t:" part:**

Each letter is an option:

* + i → input file
  + o → output file
  + t → number of threads

Colon : after each letter means: the option requires an argument. EXAMPLE



**6.1.2 getopts**

getopts is a built-in Bash utility used to parse short options (like -i,

-o, -t) and their associated arguments in a safe and consistent way.

**It's used to:**

* + Read options passed to your script.
  + Assign those values to variables.
  + Handle errors gracefully.

SYNTAX :

while getopts "abc:" opt; do case "$opt" in

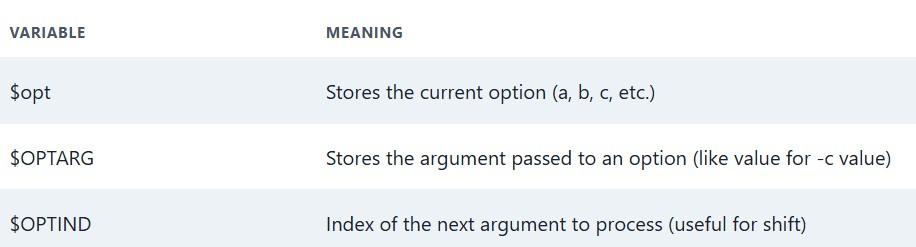
1. echo "Option A selected" ;;
2. echo "Option B selected" ;;
3. echo "Option C has value: $OPTARG" ;;

\?) echo "Invalid option: -$OPTARG"; exit 1 ;; esac

done EXPLANATION:

1. a, b are **flags** (no argument needed).
2. c: means **option -c requires an argument**.
3. \? handles **invalid options**.

## VARIABLES EXPLAINED:



**EXAMPLE CODE :**

#!/bin/bash

# Check if at least one filename is provided if [ -z "$1" ]; then

echo "procfile: No file specified" exit 1

fi

# Loop through all filenames given as arguments for filename in "$@"; do

# Strip the extension to create the intermediate PPM filename pnmfile="${filename%.\*}.ppm"

# Convert based on file extension case "$filename" in

\*.jpg )

echo "procfile: $filename is already a JPEG file" continue ;; # Skip processing

\*.tga )

tgatoppm "$filename" > "$pnmfile" ;;

\*.xpm )

xpmtoppm "$filename" > "$pnmfile" ;;

\*.pcx )

pcxtoppm "$filename" > "$pnmfile" ;;

\*.tif )

tifftopnm "$filename" > "$pnmfile" ;;

\*.gif )

giftopnm "$filename" > "$pnmfile" ;;

\* )

echo "procfile: $filename is an unknown image type" exit 1 ;;

esac

# Create output JPEG filename outfile="${pnmfile%.ppm}.new.jpg" # Convert PPM to JPEG

pnmtojpeg "$pnmfile" > "$outfile" # Remove intermediate PPM file rm "$pnmfile"

echo "Converted $filename → $outfile" done

## 6.2 Typed Variables in Bash

By default, Bash treats all variables as strings. This works fine for many scripts, but in certain cases — especially involving arithmetic or data integrity — you may want to define variables with a specific type.

That’s where typed variables come into play, using the built-in commands: declare or typeset.

declare and typeset are basically the same in Bash. typeset is used more in older shells like ksh, while declare is preferred in Bash.

# Why Do We Need Typed Variables?

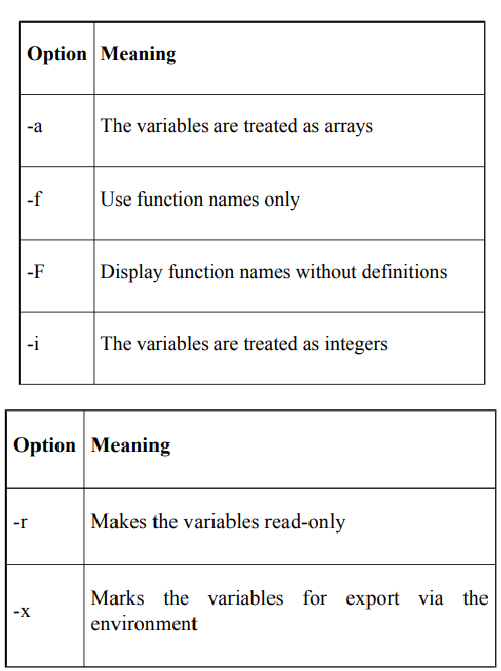
In larger scripts or scientific pipelines (like bioinformatics), you:

* Want to avoid errors caused by treating strings as numbers.
* Need arrays, integers, or read-only variables to preserve logic.
* May want to debug or optimize memory usage and behavior.

Basic Syntax:

declare -i num

This tells Bash: "Treat num as an integer — even if I try to assign it a string.



1. Integer Variable with declare -i

declare -i count count="4 + 2"

echo $count # Output: 6

Other Useful Flags with declare



**6.3 Integer Variables and Arithmetic in Bash** Bash provides built-in arithmetic features that allow scripts to perform integer-based calculations easily and efficiently. These features are

especially useful in automation, system scripting, and scientific tasks like counting, loop control, resource estimation, and more.

**1. Arithmetic Expressions**

Bash supports arithmetic evaluation using the syntax:

$((expression))

This behaves similarly to arithmetic in C or Java — performing direct evaluation of the expression. Inside the expression, you may refer to variables **with or without** a leading

$.

For example, if x=10 and y=5, then $((x + y)) gives 15.

1. ******Arithmetic Use Cases**
   * Counting iterations in loops
   * Calculating percentages or sizes (e.g., disk space, CPU time)
   * Time math like estimating days/weeks until a deadline
   * Dynamic logic such as increasing retry counts, scores, or counters

🔸 **3. Quoting Arithmetic in Strings**

When embedding arithmetic in strings, always use double quotes to avoid issues with word splitting or expansion of special characters (like ~, $).

Example: Embedding remaining weeks until New Year based on current day of the year:

"Only $(( (365 - $(date +%j)) / 7 )) weeks until the New Year"

This uses date +%j to get the current day number (1–365) and computes remaining weeks.

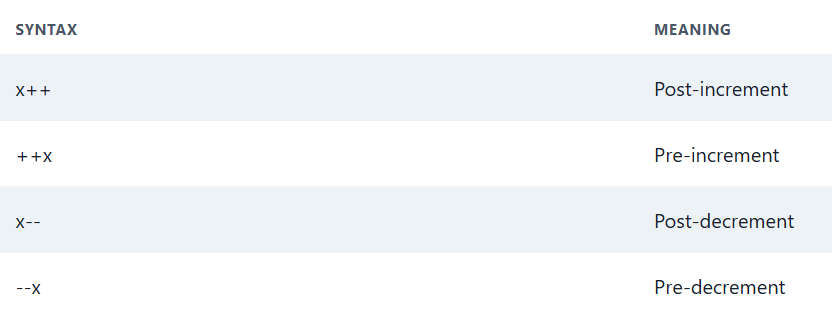
1. **Arithmetic Operators Supported**

Bash supports many common operators:



1. ******Increment & Decrement**

Bash supports the same pre/post increment and decrement as C or Java:



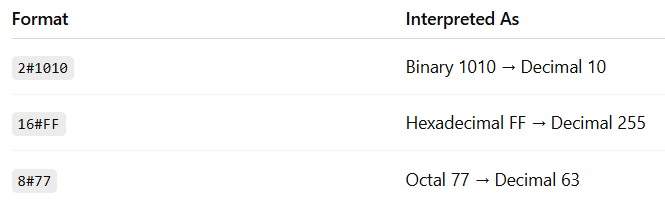
1. **Relational and Logical Operators**

Useful for condition checking, especially in arithmetic-based loops or decisions.



1. **Base Conversions**

Bash supports interpreting values from other bases:



### 6.4 Arithmetic for Loop

In Bash, the arithmetic for loop is similar to the for loop used in C, Java, and Python (range-based).

Syntax:

for (( initialization ; condition ; increment )) do

# commands to execute done

Each part:

initialization: sets a variable (only once at the start) condition: checks a condition before each iteration increment: updates the variable after each iteration Example 1: Counting from 1 to 5

for (( i=1; i<=5; i++ ))

do

echo "Number: $i" done

Output:

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

### 6.5 ARRAYS

An array in Bash is a single variable that holds multiple values, each identified by a numerical index.

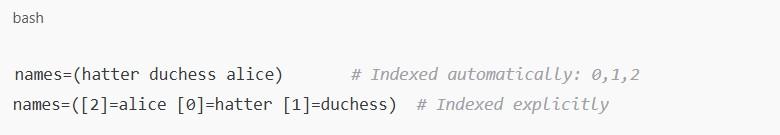
🔸 Key Concepts

* Indexing starts at 0
* Each stored value is called an element
* Arrays can hold strings, integers, or a mix of both

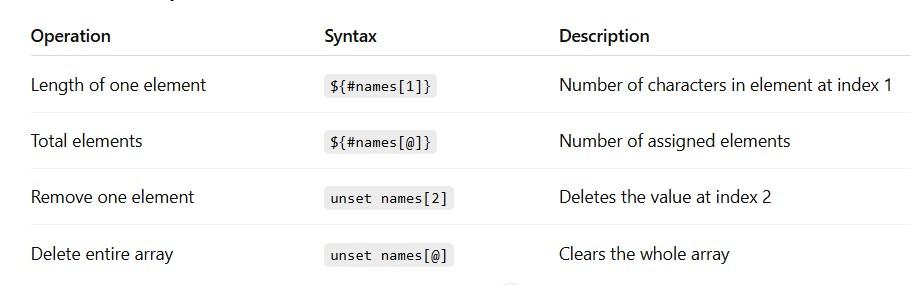
🔸 Creating and Assigning Values

**Individual Assignment:**

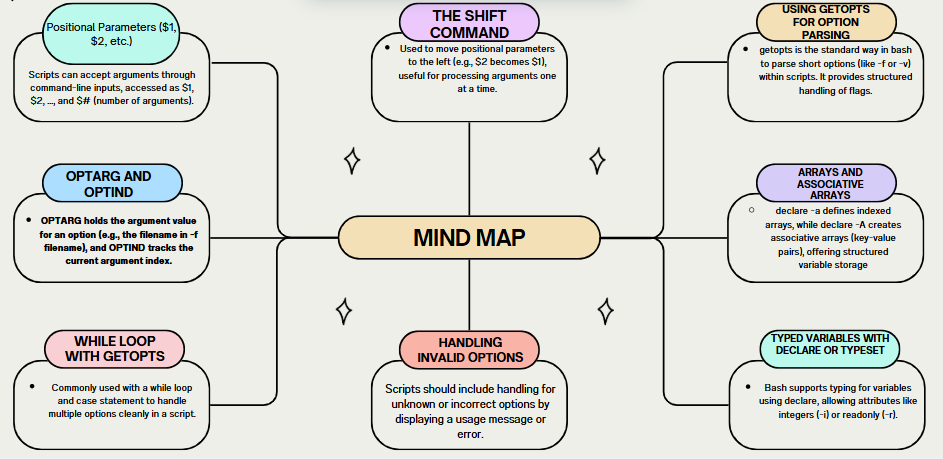


**Compound Assignment:**

**USEFUL OPERATIONS:**



**SUMMARY**

****

.