Chapter 1. Getting started

In this section, you'll learn the foundational elements of Bash scripting, including how to define and use variables, write comments, and create simple scripts. Each concept builds on the last, helping you get comfortable writing your own Bash scripts from scratch.

1.1. Variables

Variables in Bash are used to store and reference data. You assign a value using the = operator, and retrieve the value using the $ symbol.

```
NAME="TRAW"

echo ${NAME}      # => TRAW :This prints the variable value
echo $NAME        # => TRAW :This prints the variable value
echo "$NAME"      # => TRAW :This prints the variable value
echo '$NAME'      # => $NAME : This will print the exact string, not the value
echo "${NAME}!"   # => TRAW! : Concatenation of variable value with !
```

Important

When declaring variables, there must be no space between the variable name, the = sign, or the value:

```
NAME = "TRAW"    # => Error (due to spaces)
```

To **assign** a value, you don't use the dollar sign ($). But when you **reference** a variable's value, you do:

- Assignment: `NAME="TRAW"`
- Reference: `echo $NAME`

1.2. Comments

Comments help explain your code and are ignored during execution. Use the # symbol for single-line comments.

```
# This is an inline Bash comment.
```

For multi-line comments, Bash doesn't have a dedicated syntax like other languages, but you can use a `: '` block trick.

```
: '
This is a
Multi-line comment
in bash

If it looks very neat
'
```

For multi-line comments, use `: '` to open and `'` to close the block.

Here's a simple Bash script that stores a greeting and prints it to the terminal.

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

greet="world World"
echo "$greet!, I'm new here."
# => Hello world! I'm new here.
```

You can save this script to a file and run it in two ways:

```
$ chmod +x helloworld.sh
$ ./helloworld.sh

# or just
$ bash helloworld.sh
```

1.3. Special Parameters

Special parameters in Bash are predefined variables that give you information about arguments, the shell environment, and the script's execution state.

Parameter	Description
$1…$9	Parameters 1 to 9 (positional arguments)
$0	Name of the script or the shell
$1	First argument passed to the script or function
${10}	Positional parameter 10 (and beyond)
$#	Total number of positional parameters
$*	All positional parameters as a single word

Parameter	Description
$@	All positional parameters as separate words
$$	Process ID of the current shell
$-	Current shell options
$_	Last argument of the previous command
$!	PID of the last background command
$?	Exit status of the last foreground command

$* and $@ both expand all positional parameters, but behave differently when quoted. We'll explore that further in later sections.

For deeper reference, see: Bash Hackers Wiki

1.4. Functions

Functions in Bash allow you to group reusable code into named blocks. You can define them with or without the `function` keyword.

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

get_username() {
    echo "TRAW"
}

# Or using the function keyword
function get_username() {
    echo "TRAW"
}

echo "You are $(get_username)"
```

Functions can return values using `echo`, and the result can be captured via command substitution like `$(...)`.

1.5. Conditions

Conditional statements in Bash allow you to make decisions in your scripts. The `if` statement evaluates whether a condition is true, and executes the appropriate block of code.

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

string="test"

if [[ -z "$string" ]]; then
    echo "String is empty"
elif [[ -n "$string" ]]; then
    echo "String is not empty"
fi
```

- `-z` checks if a string is empty.
- `-n` checks if a string is not empty.
- Use `[[ … ]]` for more robust conditional expressions.

1.6. Command Substitution

Command substitution lets you capture the output of a command and use it as part of another command or assignment.

```bash
# => I'm TRAW
echo "I'm $(whoami)"

# Similar to:
echo "I'm `whoami`"
```

The modern syntax `$(command)` is preferred over backticks `command` because it's easier to read and allows nesting.

1.7. Brace Expansion

Brace expansion is a convenient way to generate strings or sequences in Bash.

Example	Description
`$ echo {A,B}.txt`	Outputs: A.txt B.txt
`{A,B}`	Expands to: A B
`{1..5}`	Expands to: 1 2 3 4 5

Brace expansion is useful for generating multiple similar strings without writing them out manually. Great for looping over files or ranges.

For more examples, see: Bash Hackers Wiki

1.8. Operators in Bash

Operators allow you to perform operations on values and variables in Bash. Bash provides various types of operators for different purposes:

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Ternary Operator

This article will provide a detailed explanation of each type of operator in Bash scripting.

1.8.1. Arithmetic Operators

Arithmetic operators allow you to perform mathematical operations on integers. They combine integer variables and values to produce a new integer result.

Below is a list of all the arithmetic operators available in Bash:

Operator	Description	Example
+	Addition	`$a + $b`
-	Subtraction	`$a - $b`
*	Multiplication	`$a * $b`
/	Division	`$a / $b`
%	Modulus (remainder)	`$a % $b`
**	Exponentiation (a to the power b)	`$a ** $b`

The addition, subtraction, multiplication, and division operators work as expected for mathematical operations.

The modulus (%) operator returns the remainder of the division between two integers. This is useful for checking if a number is even or odd, among other uses.

1.8.2. Assignment Operators

Assignment operators store values or the result of an expression into a variable.

Operator	Description	Example
=	Simple assignment	a=5
+=	Add and assign	a+=2
-=	Subtract and assign	a-=3
*=	Multiply and assign	a*=4
/=	Divide and assign	a/=5
%=	Modulus and assign	a%=6

Simple assignment allows storing values in a variable. The combined assignment operators like `=` and `-=` allow modifying variables more concisely. For instance, `a=b` is the same as `a = a + b`.

1.8.3. Relational Operators

Relational operators are used for comparing integers and strings to evaluate conditions.

Operator	Description	Example
-eq	Equal to	`[ $a -eq $b ]`
-ne	Not equal to	`[ $a -ne $b ]`
-gt	Greater than	`[ $a -gt $b ]`
-ge	Greater than or equal to	`[ $a -ge $b ]`
-lt	Less than	`[ $a -lt $b ]`
-le	Less than or equal to	`[ $a -le $b ]`
<	Less than (strings)	`[ "$a" < "$b" ]`
<=	Less than or equal (strings)	`[ "$a" ⍰ "$b" ]`
>	Greater than (strings)	`[ "$a" > "$b" ]`
>=	Greater than or equal (strings)	`[ "$a" >= "$b" ]`
==	Equal to	`[ $a == $b ]`
!=	Not equal to	`[ $a != $b ]`

`-eq`, `-ne`, `-gt`, `-ge`, `-lt`, and `-le` work on integers, while <, ⍰, >, and >= work on string sorting order. These are commonly used in `if` statements and loops to control program flow. The equal (==) and not equal (`!=`) operators are useful for comparing integers.

1.8.4. Logical Operators

Logical operators are used for combining and negating conditional expressions.

Operator	Description	Example
!	Negation	`! [ $a -eq $b ]`
&&	AND	`[ $a -eq 5 ] && [ $b -eq 10 ]`
||	OR	`[ $a -eq 5 ] || [ $b -eq 10 ]`

The NOT operator (`!`) inverts a true condition to false or a false condition to true. The AND operator (&&) evaluates to true if both operands are true, while the OR operator (`||`) evaluates to true if either operand is true. These allow creating complex conditional logic in scripts.

1.8.5. Bitwise Operators

Bitwise operators manipulate integers at the bit level.

Operator	Description	Example
&	Bitwise AND	`$a & $b`
|	Bitwise OR	`$a | $b`
~	Bitwise NOT	`~$a`
^	Bitwise XOR	`$a ^ $b`
<<	Left shift	`$a << 2`
>>	Right shift	`$a >> 2`

Bitwise operators treat integers as binary strings and set, unset, or toggle bits at specific positions. This allows bitmasking, toggling, and shifting values for flags and low-level binary operations.

1.8.6. Ternary Operator

The ternary operator allows simple conditional expressions.

Syntax	Description
`<condition> ? <if true> : <if false>`	Evaluates test condition and returns the "if true" or "if false" result

The ternary operator is structured as `condition ? resultIfTrue : resultIfFalse`. It tests the given condition and returns the specified result depending on whether the condition evaluated to true or false. This provides a concise way to assign values based on conditions.

Chapter 2. Bash Built-in Commands

Built-in commands are part of the shell itself and do not require launching a separate executable. Because they're compiled directly into the shell, they are generally faster and more efficient than external commands.

2.1. Built-ins

Command	Description
`.` (dot)	Same as `source`. Executes a script in the current shell environment.
`&`	Runs a job in the background.
`x`	Evaluates the arithmetic expression `x`.
`:`	No-op command. Does nothing and always succeeds (similar to `true`).
`[ t ]`	Tests the expression `t` (same as the `test` command).
`[[ e ]]`	Evaluates the conditional expression `e`. Preferred for complex conditions.
`alias`	Creates a shortcut or alias for another command.
`bg`	Resumes a suspended job in the background.
`bind`	Binds keyboard sequences to readline functions or macros.
`break`	Exits a `for`, `while`, `select`, or `until` loop early.
`builtin`	Forces execution of a shell built-in, bypassing any function or alias with the same name.

2.2. Built-ins: C–D

Command	Description
`caller`	Displays the call stack — useful inside functions and scripts.
`case`	Executes commands based on pattern matching.
`cd`	Changes the current working directory.
`command`	Runs a command while bypassing shell function or alias lookup.
`compgen`	Displays possible autocompletion options for a given word.
`complete`	Specifies how arguments should be completed for a command.
`compopt`	Sets completion behavior for arguments.

Command	Description
`continue`	Skips the rest of a loop and begins the next iteration.
`coproc`	Starts a background process connected to the script via file descriptors.
`declare`	Declares variables and assigns attributes like `-a` for arrays or `-i` for integers.
`dirs`	Shows a list of remembered directories in the directory stack.
`disown`	Removes a job from the shell's job table (won't be affected by HUP signals).

2.3. Built-ins: E–F

Command	Description
`echo`	Prints text or variables to standard output.
`enable`	Enables or disables built-in shell commands.
`eval`	Combines arguments into a single string and evaluates them as a command.
`exec`	Replaces the current shell process with a different program.
`exit`	Exits the shell with a status code.
`export`	Makes variables available to child processes.
`false`	Always returns a non-zero exit status (indicating failure).
`fc`	Lists or edits commands from history.
`fg`	Resumes a background job in the foreground.
`for`	Iterates over a list of items, executing commands for each.
`function`	Declares a shell function.

2.4. Built-ins: G–P

Command	Description
`getopts`	Parses positional parameters for flags (used in argument parsing).
`hash`	Caches the location of commands to speed up lookup.

Command	Description
`help`	Displays help information for Bash built-in commands.
`history`	Shows the command history.
`if`	Runs a block of commands conditionally.
`jobs`	Lists current background and suspended jobs.
`kill`	Sends a signal (e.g., `SIGKILL`, `SIGTERM`) to a process by PID.
`let`	Performs arithmetic evaluation on arguments.
`local`	Declares a variable with local scope inside functions.
`logout`	Exits a login shell session.
`mapfile`	Reads lines from standard input into an array.
`popd`	Removes the top directory from the directory stack.
`printf`	Prints formatted output (more powerful than `echo`).
`pushd`	Adds a directory to the directory stack.
`pwd`	Displays the current working directory.

2.5. Built-ins: R–Z and Symbols

Command	Description
`read`	Reads one line of input and assigns it to variables.
`readarray`	Reads lines into an array (same as `mapfile`).
`readonly`	Marks a variable as read-only (cannot be modified).
`return`	Exits a function and optionally returns a value.
`select`	Presents a numbered menu to the user for input.
`set`	Sets shell flags or environment variables.
`shift`	Shifts positional parameters left by one.
`shopt`	Toggles optional shell behavior (like `globstar`).
`source`	Executes commands from a file in the current shell.
`suspend`	Suspends the shell until it receives `SIGCONT`.

Command	Description
`test`	Evaluates a condition and returns success or failure.
`time`	Measures the time taken by a command to execute.
`times`	Displays shell CPU time usage.
`trap`	Specifies commands to run on receiving a signal.
`true`	Always returns a successful (0) exit status.
`type`	Tells how a command would be interpreted (alias, function, builtin, etc.).
`typeset`	Declares variables (same as `declare`, legacy use).
`ulimit`	Sets resource usage limits.
`umask`	Sets default permission bits for new files/directories.
`unalias`	Removes one or more aliases.
`unset`	Unsets variables or shell functions.
`until`	Repeats a command block until a condition becomes true.
`wait`	Waits for background jobs to finish.
`while`	Repeats a command block while a condition remains true.
`{ c; }`	Groups commands in the current shell context.

Use the `help` command to see built-in command details right from your terminal:

Or to get help for a specific built-in:

```
help builtin_name
```

Chapter 3. Bash Arrays

Arrays let you store multiple values in a single variable. Bash supports both indexed and associative arrays. Here's how to declare them:

3.1. Array Declaration Syntax

Syntax	Description
`ARRAY=()`	Declares an **indexed** array and initializes it as empty. Also used to reset an existing array.
`ARRAY[0]=value`	Sets the first element of an indexed array. If the array doesn't exist yet, it's created.
`declare -a ARRAY`	Declares an indexed array without initializing it.
`declare -A ARRAY`	Declares an **associative** array. This is the only way to create associative arrays in Bash.

3.2. Storing Values in Arrays

There are many ways to assign values to arrays in Bash. This includes assigning individual elements, using compound assignment, copying from other arrays, and even reading from files.

Syntax	Description
`ARRAY[N]=VALUE`	Sets element `N` of an **indexed** array to `VALUE`. `N` can be any valid arithmetic expression.
`ARRAY[STRING]=VALUE`	Sets element `STRING` of an **associative** array to `VALUE`.
`ARRAY=VALUE`	Assigns `VALUE` to index `0` by default. This also works for associative arrays — `VALUE` is stored under key `"0"`.
`ARRAY=(E1 E2 …)`	Compound assignment for indexed arrays. Clears the array and assigns values starting at index 0. Does not work for associative arrays.
`ARRAY=([X]=E1 [Y]=E2 …)`	Indexed array assignment using index-value pairs. `X` and `Y` are arithmetic expressions.
`ARRAY=([S1]=E1 [S2]=E2 …)`	Compound assignment for **associative** arrays using string keys.
`ARRAY+=(E1 E2 …)`	Appends elements to the existing array.

Syntax	Description
`ARRAY=("${ARRAY[@]}" E1 …)`	Alternative method for appending elements using array expansion.
`ARRAY=("${ANOTHER_ARRAY[@]}")`	Copies elements from ANOTHER_ARRAY into ARRAY. Quotes are optional.
`ARRAY=("${ARRAY[@]}" "${ANOTHER_ARRAY[@]}")`	Concatenates two arrays. You can omit the quotes or use * instead of @.
`ARRAY=($(cat file))`	Reads a file and stores each word or line (depending on IFS) into an array.
`ARRAY=({N..M})`	Uses brace expansion to populate the array. N and M can be numbers or letters.

Compound assignment is very powerful and can be used to build arrays efficiently. Just be cautious: reassigning an array this way will unset previous values unless += is used.

For clarification: When you use the subscripts @ or `for mass-expanding arrays, their behavior matches that of $@ and $` when expanding positional parameters. It's worth reading that article to fully understand the nuances.

3.3. Getting Values from Arrays

You can access specific elements, the entire array, or a slice of elements using parameter expansion.

Syntax	Description
`${ARRAY[N]}`	Gets the value at index N in an **indexed** array. If N is negative, it's counted backward from the highest assigned index (only for reading, not assigning).
`${ARRAY[S]}`	Gets the value at key S in an **associative** array.
`${ARRAY[@]}`	Expands to all elements (unquoted: expands to a word list).
`"${ARRAY[@]}"`	Expands to all elements, each quoted separately.
`${ARRAY[*]}`	Expands to all elements as a single word (if unquoted).
`"${ARRAY[*]}"`	Expands to all elements as one quoted string.

Syntax	Description
`${ARRAY[@]:N:M}`	Expands to a slice of M elements starting at index N (unquoted).
`"${ARRAY[@]:N:M}"`	Quoted slice expansion — each element quoted separately.
`${ARRAY[*]:N:M}`	Slice expansion with * (unquoted).
`"${ARRAY[*]:N:M}"`	Slice expansion with *, quoted as a single string.

For clarification: When using @ or `to expand arrays, their behavior mirrors how $@ and` $ behave with positional parameters.

- Quoted "$@" → each element separately quoted.
- Quoted "$*" → all elements quoted as one string.
- Unquoted @ and * behave the same — they expand to a space-separated list of elements.

When using slicing, the same quoting rules apply. If you're getting unexpected results, check your Bash version and test in your shell.

3.4. Indexes and Metadata

You can inspect arrays using special parameter expansion patterns that give you access to their structure and metadata.

Syntax	Description
`${ARRAY[0]}`	Gets the first element. See the **Getting Values** section.
`${ARRAY[-1]}`	Gets the last element (backward index — read-only).
`${ARRAY[*]}`	Expands to all elements. See the **Getting Values** section.
`${ARRAY[@]}`	Same as above. See the **Getting Values** section.
`${#ARRAY[@]}`	Expands to the **number of elements** in the array.
`${#ARRAY[*]}`	Same as above — total number of items.
`${#ARRAY}`	Expands to the **length** of the first element only.
`${#ARRAY[N]}`	Expands to the **length** of the string at index N.
`${#ARRAY[STRING]}`	Expands to the length of the associative array value at key STRING.
`${#ARRAY[@]:N:M}`	Expands to the number of elements from index N up to N+M. M is optional.
`${#ARRAY[*]:N:M}`	Same as above with * instead of @.

Syntax	Description
`${!ARRAY[@]}`	Expands to a list of all indexes in the array.
`${!ARRAY[*]}`	Same as above.

3.5. Destroying Arrays

You can remove individual elements or entire arrays using the `unset` built-in.

Syntax	Description
`unset -v ARRAY`	Unsets the entire array.
`unset -v ARRAY[@]`	Same — clears the entire array.
`unset -v ARRAY[*]`	Same — clears the entire array.
`unset -v 'ARRAY[N]'`	Unsets the element at index N. Always quote the index.
`unset -v ARRAY[STRING]`	Removes an element from an associative array at key `STRING`.
`ARRAY=( ${ARRAY[@]/STR*/} )`	Removes elements matching a pattern (`STR`).
`ARRAY=( ${ARRAY[]/STR/} )`	Same as above.

It's best to explicitly use `-v` with `unset` for clarity and portability.

See: Bash Hackers Wiki

3.6. Array Discussion

This section takes a deeper look at how arrays work in Bash. You'll explore different ways to assign values to both indexed and associative arrays, how to append and merge them, and how to loop through elements. We'll also cover how to pass arrays to functions, extract values, and remove elements using built-in tools and expansion syntax.

3.6.1. Compound Indexed Array Assignment

You can assign an entire array at once using parentheses. This sets all elements in one step, starting from index 0.

```
# Compound array assignment - sets the whole array
```

```
# NAMES => Creg Jan Anna
NAMES=('Creg' 'Jan' 'Anna')

# Using declare with compound assignment
declare -a Numbers=(1 2 3)
```

3.6.2. Indexed Array Assignment

You can also assign elements one at a time by specifying the index. The index can be any valid arithmetic expression.

```
NAMES[0]='Creg'
NAMES[1]='Jan'
NAMES[2]='Anna'
# NAMES => Creg Jan Anna
```

3.6.3. Indexed Array Assignment Using Brace Expansion

Brace expansion is a quick way to generate predictable patterns for array elements.

```
ARRAY1=(foo{1..2})        # => foo1 foo2
ARRAY2=({A..D})           # => A B C D
ARRAY3=({1..5})           # => 1 2 3 4 5
ARRAY4=({A..B}{1..2})     # => A1 A2 B1 B2
```

3.6.4. Key-Based Associative Array Assignment

Associative arrays store values using string keys. You must use `declare -A` to define one.

```
declare -A person

person[name]="Jay"
person[age]=22
person[eye_color]="blue"

# person => Jay 22 blue
```

3.6.5. Compound Associative Array Assignment

You can initialize all keys and values at once using a compound assignment.

```
: '
Individual mass-setting for associative arrays.
The named indexes (e.g., name, age) are strings.
'

declare -A person  # Required declaration
person=(["name"]="Jay" [age]=22 ["eye_color"]="blue")

# person => Jay 22 blue
```

3.6.6. Working with Associative Arrays

Once an associative array is created, you can interact with it using standard Bash expansions.

```
declare -A person
person=(["name"]="Jay" [age]=22 ["eye_color"]="blue")

echo ${person[name]}    # Get person's name
echo ${person[@]}       # Get all values
echo ${person[*]}       # Get all values
echo ${!person[@]}      # Get all keys
echo ${#person[@]}      # Get number of elements

unset -v person[age]    # Delete age entry
```

All of these operations work the same way with indexed arrays — just use numeric indexes instead of string keys.

3.6.7. Appending Values to Arrays

You can add new elements to an array using the += operator or by expanding the current array into a new list.

```
NAMES=('Creg' 'Jan' 'Anna')

NAMES+=('Jay' 'Jimmy' 'Tom')
# NAMES => Creg Jan Anna Jay Jimmy Tom

# Same as:
NAMES=(${NAMES[@]} 'Jay' 'Jimmy' 'Tom')
# NAMES => Creg Jan Anna Jay Jimmy Tom
```

3.6.8. Iterating Over an Indexed Array

You can loop through array values directly or use indices to access each element.

```
names=('Jay' 'Joe' 'Jimmy')

# Direct element access
for name in "${names[@]}"; do
    echo $name
done

# Index-based access
for i in "${!names[@]}"; do
    printf "%s\t%s\n" "$i" "${names[$i]}"
done
```

3.6.9. Iterating Over an Associative Array

Looping over associative arrays is similar — just replace numeric indexes with string keys.

```
declare -A person
person=(["name"]="Jay" [age]=22 ["eye_color"]="blue")

# Loop over values
for value in "${person[@]}"; do
    echo $value
done

# Loop over keys and values
for key in "${!person[@]}"; do
    echo "$key  ${person[$key]}"
done
```

3.6.10. Merging / Concatenating Arrays

You can merge multiple arrays into one using parameter expansion.

```
ARRAY1=(foo{1..2})   # => foo1 foo2
ARRAY2=({A..D})      # => A B C D
ARRAY3=({1..5})      # => 1 2 3 4 5

# Merge all into one
ARRAY4=(${ARRAY1[@]} ${ARRAY2[@]} ${ARRAY3[@]})
# ARRAY4 => foo1 foo2 A B C D 1 2 3 4 5
```

```
# You can also use * instead of @
# ARRAY4=(${ARRAY1[*]} ${ARRAY2[*]} ${ARRAY3[*]})
```

3.6.11. Array Arguments to Functions

You can pass arrays to functions using **namerefs** (`local -n`) to make them accessible by name.

```
function extract()
{
    # Create a local reference to the array passed as $1
    local -n local_names=$1
    local index=$2
    echo "${local_names[$index]}"
}

names=('Jay' 'Joe' 'Jimmy')

extract names 2     # => Jimmy
```

Namerefs (using `local -n`) are supported in Bash 4.3 and later.