

Built-in Shell Commands

LinuXpert Systems

No.5, Dr. Ambedkar St, Rathinapuram,

Nanganallur, Chennai – 600 061.

Phone: 98841 65649, 90808 01382

E-mail: baskar@linuxpert.in

TABLE OF CONTENTS

1. alias 1
2. bg 7
3. cd 11
4. echo 18
5. eval 25
6. exit 30
7. export 35
8. fg 41
9. history 45
10. jobs 51
11. kill 58
12. logout 63
13. pwd 67
14. read 72
15. set 77
16. source 84
17. type 90
18. ulimit 96
19. umask 103
20. unalias 109
21. unset 114

1. alias

The `alias` command is a Bash built-in used to **create shortcuts (nicknames)** for long or frequently used commands. Aliases exist only for the current shell session unless you save them in a shell configuration file.

Syntax

```
alias name='command'
```

- `name` – the shortcut you want to create.
- `command` – the command that will run when you type the shortcut.

Basic Example

Create an alias named `ll` for `ls -l`:

```
alias ll='ls -l'
```

Now typing:

```
ll
```

is equivalent to:

```
ls -l
```

More Examples

1. Long Listing with Hidden Files

```
alias la='ls -la'
```

Usage:

```
la
```

2. Colored Output

```
alias ls='ls --color=auto'
```

Now:

```
ls
```

automatically displays colored output.

---

3. Go Back One Directory

```
alias ..='cd ..'
```

Usage:

```
..
```

---

4. Go Back Two Directories

```
alias ...='cd ../..'
```

---

5. Update Packages (Ubuntu)

```
alias update='sudo apt update && sudo apt upgrade'
```

Usage:

```
update
```

---

6. Prevent Accidental File Deletion

```
alias rm='rm -i'
```

Now:

```
rm file.txt
```

asks for confirmation before deleting.

7. Copy with Confirmation

```
alias cp='cp -i'
```

---

8. Move with Confirmation

```
alias mv='mv -i'
```

Display All Aliases

```
alias
```

Example output:

```
alias ll='ls -l'
alias la='ls -la'
alias rm='rm -i'
```

Display a Specific Alias

```
alias ll
```

Output:

```
alias ll='ls -l'
```

Remove an Alias

Use the `unalias` command.

```
unalias ll
```

Remove all aliases:

```
unalias -a
```

Make an Alias Permanent

Temporary aliases disappear when you close the terminal.

To make them permanent, add them to:

For Bash:

```
~/.bashrc
```

Example:

```
alias ll='ls -l'
alias la='ls -la'
alias rm='rm -i'
```

Then reload the file:

```
source ~/.bashrc
```

Check Whether a Command Is an Alias

```
type ll
```

Output:

```
ll is aliased to `ls -l'
```

Override an Alias Temporarily

Suppose:

```
alias rm='rm -i'
```

To use the original `rm` command once:

```
\rm file.txt
```

or

```
command rm file.txt
```

Advantages

- Saves typing time.
- Makes long commands easier to remember.
- Helps prevent accidental mistakes (e.g., `rm -i`).
- Improves productivity for frequently used commands.

Limitations

- Aliases work only in the current shell unless saved.
- They cannot accept parameters like functions can.
- They are expanded only for interactive shells by default.

Commonly Used Aliases

Alias	Expands To	Purpose
`ll`	`ls -l`	Long directory listing
`la`	`ls -la`	Show all files including hidden
`l`	`ls -CF`	Classify directory contents
`..`	`cd ..`	Go up one directory
`...`	`cd ../..`	Go up two directories
`grep`	`grep --color=auto`	Highlight matched text
`rm`	`rm -i`	Ask before deleting
`cp`	`cp -i`	Ask before overwriting
`mv`	`mv -i`	Ask before overwriting
`cls`	`clear`	Clear the terminal screen

Summary

- **Command:** `alias`
- **Purpose:** Creates a shortcut for another command.
- **View all aliases:** `alias`
- **View one alias:** `alias name`
- **Create alias:** `alias name='command'`
- **Remove alias:** `unalias name`
- **Permanent aliases:** Add them to `~/.bashrc` and run `source ~/.bashrc`.

The `alias` command is one of the simplest yet most useful Bash features for customizing your command-line environment and reducing repetitive typing.

2. bg

The `bg` command is used to **resume a stopped job and run it in the background**. It is one of the Bash job control commands.

Syntax

```
bg [j ob_spec]
```

- `job_spec` is the job number (for example, `%1`, `%2`).
- If no job is specified, `bg` resumes the current stopped job.

How `bg` Works

When you start a command in the foreground, it occupies the terminal until it finishes.

If you suspend it by pressing **Ctrl + Z**, the process is **stopped** (paused).

The `bg` command restarts the stopped process and lets it continue running **in the background**, allowing you to use the terminal for other commands.

Example 1: Suspend and Resume in Background

Start a long-running command:

```
sleep 100
```

Press:

```
Ctrl + Z
```

Output:

```
[1]+  Stopped                 sleep 100
```

Now resume it in the background:

```
bg
```

Output:

```
[1]+ sleep 100 &
```

The process continues running in the background.

Example 2: Resume a Specific Job

Start multiple jobs.

```
sleep 100
```

Press **Ctrl** + **Z**

Start another command:

```
sleep 200
```

Press **Ctrl** + **Z**

View jobs:

```
jobs
```

Output:

```
[1]-  Stopped  sleep 100
[2]+  Stopped  sleep 200
```

Resume Job 1:

```
bg %1
```

Resume Job 2:

```
bg %2
```

Example 3: Background a Text Editor

Open a file:

```
nano file.txt
```

Press:

```
Ctrl + Z
```

Output:

```
[1]+ Stopped nano file.txt
```

Run:

```
bg
```

The editor continues running in the background.

To bring it back to the foreground:

```
fg
```

Checking Background Jobs

Use:

```
jobs
```

Example:

```
[1]- Running  sleep 100 &
[2]+ Running  sleep 200 &
```

Difference Between & and bg

Feature	&	bg
Starts a command in the background		
Resumes a stopped command		
Requires Ctrl + Z first		

Example:

Start directly in the background:

```
sleep 100 &
```

Resume an already stopped job:

```
bg
```

Related Commands

Command	Purpose
`jobs`	List current jobs
`fg`	Bring a background/stopped job to the foreground
`bg`	Resume a stopped job in the background
`kill`	Terminate a job or process
`Ctrl + Z`	Suspend the current foreground process
`Ctrl + C`	Terminate the current foreground process

Common Error

If you run:

```
bg
```

without any stopped jobs, you'll see:

```
bash: bg: current: no such job
```

This means there are no suspended jobs available to resume.

Summary

- `bg` resumes a **stopped** job and runs it in the **background**.
- Suspend a foreground job with **Ctrl + Z**.
- Use `jobs` to see available jobs.
- Resume the current job with:

  ```
  bg
  ```

- Resume a specific job with:

  ```
  bg %1
  ```

- Use `fg` to bring a background job back to the foreground.

3. cd

The `cd` (**Change Directory**) command is used to **navigate between directories (folders)** in a Linux or Unix shell. It is one of the most commonly used shell built-in commands.

Syntax

```
cd [directory]
```

- `directory` – The path of the directory you want to move to.
- If no directory is specified, `cd` takes you to your home directory.

Basic Examples

1. Go to your home directory

```
cd
```

or

```
cd ~
```

Example

```
$ pwd
/etc

$ cd

$ pwd
/home/bharath
```

---

2. Change to an absolute path

An **absolute path** starts from the root (`/`).

```
cd /etc
```

Example:

```
$ cd /var/log
```

3. Change to a relative path

A **relative path** starts from your current directory.

Current directory:

```
/home/bharath
```

Command:

```
cd Documents
```

Moves to:

```
/home/bharath/Documents
```

---

4. Go up one directory

```
cd ..
```

Example:

Current:

```
/home/bharath/Documents
```

After:

```
cd ..
```

Result:

```
/home/bharath
```

5. Go up two directories

```
cd ../..
```

Example:

Current:

```
/home/bharath/Documents/Linux
```

Result:

```
/home/bharath
```

---

6. Go to the previous directory

```
cd -
```

Example:

```
$ pwd
/home/bharath

$ cd /etc

$ pwd
/etc

$ cd -
/home/bharath

$ pwd
/home/bharath
```

Useful for quickly switching between two directories.

7. Go to the root directory

```
cd /
```

Result:

```
/
```

---

8. Go to a user's home directory

```
cd /home/username
```

or

```
cd ~username
```

Example:

```
cd ~john
```

Moves to:

```
/home/john
```

(if the user exists).

Special Directory Symbols

Symbol	Meaning
`.`	Current directory
`..`	Parent directory
`~`	Current user's home directory
`-`	Previous working directory
`/`	Root directory

Using `cd` with Spaces in Directory Names

If a directory name contains spaces:

```
cd "My Folder"
```

or

```
cd My\ Folder
```

Verify the Current Directory

Use:

```
pwd
```

Example:

```
$ cd /usr/local
```

```
$ pwd
/usr/local
```

Common Errors

Directory does not exist

```
$ cd abc
```

```
bash: cd: abc: No such file or directory
```

Permission denied

```
$ cd /root
```

```
bash: cd: /root: Permission denied
```

Occurs when you don't have permission to access the directory.

Why `cd` is a Shell Built-in

`cd` changes the **current working directory of the shell process itself**. If it were an external program, it could only change its own directory, and after it exited, your shell would remain in the original directory. That's why `cd` is implemented as a **shell built-in command**.

You can verify this:

```
type cd
```

Output:

```
cd is a shell builtin
```

Useful `cd` Shortcuts

Command	Description
`cd`	Go to your home directory
`cd ~`	Go to your home directory
`cd /`	Go to the root directory
`cd ..`	Go to the parent directory
`cd ../..`	Go up two directory levels
`cd -`	Return to the previous directory
`cd /etc`	Change to an absolute path
`cd Documents`	Change to a relative path
`cd "My Folder"`	Enter a directory with spaces in its name

Exit Status

- **0** – Directory change was successful.
- **Non-zero** – An error occurred (for example, the directory doesn't exist or you don't have permission).

Example:

```
$ cd /etc
$ echo $?
0

$ cd /does_not_exist
bash: cd: /does_not_exist: No such file or directory

$ echo $?
1
```

`cd` is one of the first commands every Linux user learns because almost every task—creating files, editing configuration files, running programs, or managing directories—starts with navigating to the correct location.

4. echo

The `echo` command is used to **display text, variables, or the values of expressions on the terminal**. It is one of the most commonly used Bash built-in commands.

Syntax

```
echo [OPTION]... [STRING]...
```

- **OPTION** – Modifies the behavior of `echo`.
- **STRING** – The text or variables to display.

Basic Examples

1. Print a Message

```
echo "Hello, World!"
```

Output:

```
Hello, World!
```

---

2. Print Multiple Words

```
echo Linux is awesome
```

Output:

```
Linux is awesome
```

---

3. Print Variable Values

```
name="John"
echo $name
```

Output:

```
John
```

or

```
echo "My name is $name"
```

Output:

```
My name is John
```

---

4. Print Environment Variables

```
echo $HOME
```

Output

```
/home/user
```

```
echo $PATH
```

Displays the system's PATH variable.

Common Options

`-n` (No Newline)

Normally, `echo` prints a newline after the output.

```
echo -n "Hello"
echo " World"
```

Output

```
Hello World
```

Without `-n`, the output would be:

```
Hello
 World
```

`-e` (Enable Escape Sequences)

Allows special escape characters.

```
echo -e "Line1\nLine2"
```

Output

```
Line1
Line2
```

Common Escape Sequences

Escape	Meaning
`\n`	New line
`\t`	Horizontal tab
`\\`	Backslash (\)
`\"`	Double quote
`\a`	Alert (bell)
`\b`	Backspace
`\r`	Carriage return
`\v`	Vertical tab

Example

```
echo -e "Name:\tJohn"
```

Output:

```
Name:   John
```

Printing Quotes

```
echo "\"Linux\""
```

Output:

```
"Linux"
```

Printing Special Characters

```
echo '$HOME'
```

Output:

```
$HOME
```

Single quotes prevent variable expansion.

Command Substitution

```
echo "Today is $(date)"
```

Example Output:

```
Today is Sun Jul 19 06:30:15 IST 2026
```

Arithmetic Expansion

```
echo $((10 + 20))
```

Output:

```
30
```

Redirecting Output to a File

Create or overwrite a file:

```
echo "Hello Linux" > file.txt
```

Append to a file:

```
echo "Second line" >> file.txt
```

Using `echo` in Shell Scripts

```
#!/bin/bash

echo "Starting backup..."
echo "Backup completed successfully."
```

Displaying Multiple Variables

```
name="Alice"
age=25

echo "Name: $name"
echo "Age: $age"
```

Output:

```
Name: Alice
Age: 25
```

Showing Exit Status

```
ls
echo $?
```

If `ls` succeeds:

```
0
```

If the previous command fails, a non-zero exit status is displayed.

Practical Examples

Welcome Message

```
echo "Welcome to Ubuntu!"
```

Display Current User

```
echo "Current user: $USER"
```

Display Current Directory

```
echo "Working directory: $(pwd)"
```

Create a Configuration File

```
echo "PORT=8080" > config.env
echo "HOST=localhost" >> config.env
```

Contents of `config.env`:

```
PORT=8080
HOST=localhost
```

Exit Status

- **0** – Command executed successfully.
- Non-zero values are uncommon for `echo` and usually indicate an implementation-specific issue.

Summary

Feature	Description
Purpose	Display text, variables, and command output
Syntax	`echo [OPTION] [STRING]`
`-n`	Do not print a trailing newline
`-e`	Interpret escape sequences like `\n` and `\t`
Common Uses	Printing messages, debugging scripts, displaying variables, writing to files, formatting output

`echo` is one of the simplest and most frequently used Bash commands. It is essential for displaying information in shell scripts, debugging, creating files, and showing variable or command output.

5. eval

`eval` is a **Bash built-in command** that takes one or more arguments, combines them into a single command, and then executes that command as if you had typed it directly into the shell.

Syntax

```
eval [arguments]
```

How `eval` Works

Normally, Bash interprets a command once.

With `eval`, Bash performs **an extra round of parsing and expansion** before executing the command.

Process:

1. Concatenate all arguments into one string.
2. Parse the string as a shell command.
3. Execute the resulting command.

Example 1: Basic Usage

```
command="ls -l"

eval "$command"
```

Output:

```
total 20
-rw-r--r-- file1
-rw-r--r-- file2
...
```

Without `eval`, Bash would just treat the variable as text.

Example 2: Running a Command Stored in a Variable

```
cmd="date"

eval "$cmd"
```

Output:

```
Sat Jul 18 10:30:45 IST 2026
```

Example 3: Multiple Commands

```
commands="pwd; whoami"

eval "$commands"
```

Output:

```
/home/bharath
bharath
```

Example 4: Variable Indirection

```
name="USER"
eval echo \$$name
```

Explanation:

```
$name      → USER
\$$name    → $USER
eval       → echo $USER
```

Output:

```
bharath
```

Without `eval`:

```
echo \$$name
```

Output:

```
$USER
```

Example 5: Creating Variables Dynamically

```
varname="student"
value="John"

eval "$varname='$value'"

echo $student
```

Output:

```
John
```

Example 6: Dynamic Command Construction

```
file="test.txt"

eval "cat $file"
```

Equivalent to:

```
cat test.txt
```

Example 7: Loop Example

```
for i in 1 2 3
do
    eval echo Number_$i
done
```

Output:

```
Number_1
Number_2
Number_3
```

Why Use `eval`?

It is useful when:

- Executing commands stored in variables.
- Generating commands dynamically.
- Performing variable indirection.
- Advanced shell scripting.
- Creating dynamic variable names.

Why is `eval` Dangerous?

`eval` executes **whatever text it receives**.

Example:

```
user_input="rm -rf /"
eval "$user_input"
```

This would execute:

```
rm -rf /
```

If the input comes from an untrusted user, this can lead to **command injection**, where arbitrary commands are executed. For this reason, avoid using `eval` with external or user-supplied input unless it has been thoroughly validated.

Check if `eval` is Built-in

```
type eval
```

Output:

```
eval is a shell builtin
```

or

```
help eval
```

Output:

```
eval: eval [arg ...]

Execute arguments as a shell command.

Combine ARGs into a single string, use the result as input to
the shell, and execute the resulting commands.

Exit Status:
Returns exit status of command or success if command is null.
```

Summary

Feature	Description
Command Type	Bash built-in
Purpose	Execute a dynamically constructed command
Syntax	`eval [arguments]`
Main Use	Run commands stored as strings or build commands dynamically
Performs	An additional round of shell parsing and expansion
Risk	Can execute unintended or malicious commands if given untrusted input

In modern Bash scripting, `eval` should be used sparingly. Many tasks that once relied on `eval` can now be handled more safely using arrays, parameter expansion, or indirect variable expansion (`${!var}`), which reduce the risk of command injection.

6. exit

The `exit` command is a **shell built-in command** used to **terminate the current shell session or shell script**. When you use `exit`, the shell closes and returns control to its parent process (or logs you out if it is your login shell).

Syntax

```
exit
```

or

```
exit [status]
```

- `status` is an optional exit status code (0–255).

Purpose

- Exit the current terminal shell.
- End a shell script.
- Return an exit status to the calling process.

Exit Status

The exit status tells the parent process whether the command or script completed successfully.

Exit Code	Meaning
`0`	Success
Non-zero	Error or specific condition

Example:

```
exit 0
```

Output:

```
(Current shell closes)
```

This indicates successful completion.

Example:

```
exit 1
```

This exits the shell or script with an error status of `1`.

Examples

Example 1: Exit the terminal

```
$ exit
logout
```

or

```
$ exit
```

The terminal session ends.

Example 2: Exit a shell script successfully

```
#!/bin/bash

echo "Backup completed successfully."
exit 0
```

Output:

```
Backup completed successfully.
```

The script finishes and returns success (`0`).

Example 3: Exit a script with an error

```
#!/bin/bash

if [ ! -f data.txt ]; then
    echo "File not found"
    exit 1
fi
```

Output (if `data.txt` does not exist):

```
File not found
```

The script exits immediately with status `1`.

Example 4: Check the exit status

Run a script:

```
./myscript.sh
```

Check its exit status:

```
echo $?
```

Output:

```
0
```

or

```
1
```

- `0` means the script completed successfully.
- Any non-zero value indicates an error or another condition.

Example 5: Exit from a nested shell

Start another shell:

```
bash
```

Prompt changes because you're now in a child shell.

Exit it:

```
exit
```

You return to the parent shell.

Using `exit` in Functions

```
#!/bin/bash

echo "Starting..."
exit 0

echo "This line will never execute."
```

Output:

```
Starting...
```

The script stops immediately after `exit`.

Difference Between `exit` and `logout`

Command	Purpose
`exit`	Exits the current shell (works in interactive shells and scripts).
`logout`	Logs out of a **login shell**. It usually doesn't work in non-login shells.

Common Uses

- End a shell session.
- Stop a script when an error occurs.
- Return success or failure status to another program.
- Exit nested shell environments.

Important Notes

- `exit` closes the **current shell**. If you run it in your main terminal, that terminal session ends.

- In shell scripts, `exit` immediately terminates the script.

- Exit codes range from **0 to 255**.
- By convention:
    - `0` = Success
    - Non-zero = Error or exceptional condition

Summary

Feature	Description
Command	`exit`
Type	Bash shell built-in
Purpose	Exit the current shell or script
Syntax	`exit [status]`
Default exit code	Status of the last executed command
Common success code	`0`
Common error code	`1`

The `exit` command is one of the most fundamental shell commands, used to terminate shells and scripts while optionally returning a status code that indicates success or failure.

7. export

The `export` command is a **Bash built-in** used to **make shell variables available to child processes (subshells and programs)**.

Syntax

```
export VARIABLE
```

or

```
export VARIABLE=value
```

or export multiple variables:

```
export VAR1=value1 VAR2=value2
```

Why is `export` needed?

When you create a variable normally,

```
NAME="Bharath"
```

it is only available in the current shell.

Example:

```
NAME="Bharath"
bash
echo $NAME
```

Output:

The variable is empty because the new Bash shell does not inherit it.

If you export it:

```
export NAME="Bharath"
bash
echo $NAME
```

Output

```
Bharath
```

The child shell inherits the variable.

Example 1

```
export CITY="Chennai"
```

```
echo $CITY
```

Output

```
Chennai
```

Example 2

Running a script

Suppose

```
export NAME="Linux"
```

Script:

```
#!/bin/bash
echo "Hello $NAME"
```

Run it

```
./script.sh
```

Output

```
Hello Linux
```

Without `export`, the script would not see the variable (unless it was sourced).

Example 3

Exporting PATH

```
export PATH=$PATH:/home/user/bin
```

This adds `/home/user/bin` to the executable search path.

Now any executable inside

```
/home/user/bin
```

can be run directly.

Example 4

Multiple variables

```
export USERNAME=bob PASSWORD=12345
```

Display exported variables

```
export
```

or

```
export -p
```

Example output

```
declare -x HOME="/home/bharath"
declare -x PATH="/usr/local/bin:/usr/bin"
declare -x SHELL="/bin/bash"
```

Here:

- `declare` → Bash declaration
- `-x` → exported variable

Export an existing variable

```
EDITOR=vim
export EDITOR
```

Equivalent to

```
export EDITOR=vim
```

Remove exported status

`export` itself cannot un-export a variable's value, but you can remove its exported attribute using:

```
export -n VARIABLE
```

Example

```
export NAME="Bharath"
export -n NAME
```

The variable still exists in the current shell, but it is no longer inherited by child processes.

Unset a variable completely

```
unset NAME
```

Check if a variable is exported

```
export -p | grep NAME
```

or

```
declare -xp NAME
```

Common environment variables

Variable	Purpose
PATH	Directories searched for executable commands
HOME	User's home directory
USER	Current username
SHELL	Current login shell
PWD	Current working directory
LANG	Language and locale settings
TERM	Terminal type
EDITOR	Default text editor

Options

Option	Description
-p	Display all exported variables
-n	Remove the export attribute from a variable
-f	Export shell functions (Bash-specific)

Example:

```
myfunc() {
    echo "Hello"
}

export -f myfunc
bash
myfunc
```

Output:

```
Hello
```

Practical example

```
$ PROJECT=digimat

$ echo $PROJECT
digimat

$ bash
$ echo $PROJECT
```

The child shell cannot see the variable because it was not exported.

Now:

```
$ export PROJECT=digimat
$ bash
$ echo $PROJECT
digimat
```

The child shell inherits the exported variable.

Summary

Command	Description
`export VAR=value`	Create and export a variable
`export VAR`	Export an existing variable
`export`	List exported variables
`export -p`	Print exported variables in reusable form
`export -n VAR`	Remove the export attribute
`export -f func`	Export a shell function (Bash)
`unset VAR`	Remove a variable completely

Key point

- A **shell variable** exists only in the current shell.
- An **exported variable** becomes an **environment variable**, making it available to child shells and programs launched from the current shell.

8. fg

The `fg` command is a **Bash shell built-in** used to **bring a background or stopped job to the foreground**. Once the job is in the foreground, you can interact with it directly from the terminal.

Syntax

```
fg
```

or

```
fg %job_number
```

Purpose

When you have a process running in the background (using `&` or the `bg` command), `fg` brings it back to the foreground so you can:

- View its output directly.
- Provide keyboard input.
- Stop it with `Ctrl + C`.
- Suspend it again with `Ctrl + Z`.

How it Works

1. Start a program.
2. Suspend it using `Ctrl + Z`.
3. Resume it in the background using `bg`.
4. Bring it back using `fg`.

Examples

Example 1: Bring the current background job to the foreground

```
sleep 100 &
```

Output:

```
[1] 12345
```

Bring it to the foreground:

```
fg
```

Output:

```
sleep 100
```

The process now occupies your terminal.

Example 2: Bring a specific job to the foreground

Start two jobs:

```
sleep 100 &
sleep 200 &
```

Check the jobs:

```
jobs
```

Output:

```
[1]- Running    sleep 100 &
[2]+ Running    sleep 200 &
```

Bring Job 1 to the foreground:

```
fg %1
```

Output:

```
sleep 100
```

Example 3: Suspend and resume

Start:

```
ping google.com
```

Suspend it:

```
Ctrl + Z
```

Output:

```
[1]+ Stopped    ping google.com
```

Continue in the background:

```
bg
```

Output:

```
[1]+ ping google.com &
```

Bring it back:

```
fg
```

Output:

```
ping google.com
```

Job Specifiers

Command	Meaning
`fg`	Bring the current job to the foreground.
`fg %1`	Bring job number 1 to the foreground.
`fg %2`	Bring job number 2 to the foreground.
`fg %+`	Bring the current (most recent) job to the foreground.
`fg %-`	Bring the previous job to the foreground.

Related Commands

Command	Purpose
`bg`	Resume a stopped job in the background.
`jobs`	List all background and stopped jobs.
`kill`	Terminate a job or process.
`Ctrl + Z`	Suspend the current foreground process.
`Ctrl + C`	Terminate the current foreground process.

Notes

- `fg` only works with **jobs created in the current shell session**.
- It cannot bring arbitrary system processes to the foreground.
- If no job number is specified, `fg` resumes the **current job** (marked with `+` in the `jobs` output).
- `fg` is available in Bash and other shells that support job control (such as `zsh` and `ksh`).

Quick Demonstration

```
$ sleep 300
^Z
[1]+  Stopped     sleep 300

$ bg
[1]+ sleep 300 &

$ jobs
[1]+ Running      sleep 300 &

$ fg
sleep 300
```

Here, `sleep 300` is first suspended with `Ctrl + Z`, resumed in the background with `bg`, and then brought back to the foreground with `fg`.

9. history

The `history` command in Linux displays the list of commands you have previously executed in the current shell. It is a Bash built-in command that helps you review, reuse, and manage your command history.

Syntax

```
history [options] [number]
```

Basic Usage

Display all previous commands

```
history
```

Example Output:

```
  1  pwd
  2  ls
  3  cd Documents
  4  mkdir test
  5  cd test
  6  touch file.txt
  7  history
```

The first column is the **history number**, followed by the command that was executed.

Display the last N commands

```
history 10
```

Displays the last 10 commands.

Example:

```
$ history 5
  18  ls
  19  pwd
  20  whoami
  21  date
  22  history 5
```

Re-run a Command from History

Execute a command by its history number

```
!15
```

Runs command number **15**.

Example:

```
$ history
 10 ls
 11 pwd
 12 date

$ !12
date
Sat Jul 18 10:30:15 IST 2026
```

Execute the previous command

```
!!
```

Example:

```
$ ls
Documents  Downloads

$ !!
ls
Documents  Downloads
```

Execute the last command beginning with a string

```
!cd
```

Runs the most recent command that starts with `cd`.

Example:

```
$ !cd
cd /home/user/Documents
```

Search Command History

Using `history` with `grep`

```
history | grep ssh
```

Example Output:

```
 25 ssh server1
 48 ssh root@192.168.1.100
```

Interactive Search

Press:

`Ctrl + R`

Type part of a previous command.

Example:

```
(reverse-i-search)`git': git status
```

Press **Enter** to execute it or use the arrow keys to edit it first.

Clear History

Clear current session history

```
history -c
```

This removes the history from the current shell's memory.

Clear and save the empty history file

```
history -c
history -w
```

This also overwrites the history file (typically `~/.bash_history`) with the cleared history.

Save Current History

Write the current shell history to the history file:

```
history -w
```

Read History from File

Reload the history file into the current shell:

```
history -r
```

Append Current Session to History File

```
history -a
```

Useful when you have multiple terminal windows open.

Delete a Specific History Entry

Delete command number 25:

```
history -d 25
```

Common Options

Option	Description
`history`	Show all history
`history N`	Show last N commands
`history -c`	Clear current history
`history -d N`	Delete history entry N
`history -w`	Write history to file
`history -r`	Read history from file
`history -a`	Append new commands to history file
`history -n`	Read new lines added to the history file by other sessions

History File

By default, Bash stores command history in:

`~/.bash_history`

View it with:

```
cat ~/.bash_history
```

Useful Environment Variables

Variable	Description
`HISTSIZE`	Number of commands kept in memory
`HISTFILESIZE`	Maximum number of commands stored in `~/.bash_history`
`HISTFILE`	Path to the history file
`HISTCONTROL`	Controls which commands are saved (e.g., ignore duplicates or commands starting with a space)

Check their values:

```
echo $HISTSIZE
echo $HISTFILESIZE
echo $HISTFILE
echo $HISTCONTROL
```

Practical Examples

```
# Show all history
history

# Show last 20 commands
history 20

# Find commands containing "apt"
history | grep apt

# Execute command number 45
!45

# Execute the previous command
!!

# Execute the last command starting with "sudo"
!sudo
```

```
# Delete history entry 12
history -d 12

# Clear history
history -c

# Save history to file
history -w

# Reload history from file
history -r
```

Summary

The `history` command is one of the most useful Bash built-ins for improving productivity. It allows you to:

- View previously executed commands.
- Search for old commands.
- Re-execute commands quickly using history expansion (`!`).
- Delete or clear history entries.
- Save and reload command history.
- Manage the shell's command history across terminal sessions.

10. jobs

The **`jobs`** command displays the status of jobs (processes) that are running or stopped in the **current shell session**. It is mainly used for **job control** when working with background (&) and suspended (`Ctrl + Z`) processes.

Syntax

```
jobs [options] [job_id]
```

What is a Job?

A **job** is a command or process that is started from the current shell.

For example:

```
sleep 300 &
```

Output:

```
[1] 12345
```

- **1** → Job number
- **12345** → Process ID (PID)

Now if you run:

```
jobs
```

Output:

```
[1]+  Running    sleep 300 &
```

Example 1 – Background Process

Start a background process:

```
sleep 100 &
```

Check jobs:

```
jobs
```

Output:

```
[1]+  Running    sleep 100 &
```

Example 2 – Suspend a Process

Run:

```
sleep 100
```

Press:

```
Ctrl + Z
```

Output:

```
^Z
[1]+  Stopped    sleep 100
```

Now:

```
jobs
```

Output:

```
[1]+  Stopped    sleep 100
```

Resume it in the background:

```
bg %1
```

Check again:

```
jobs
```

Output:

```
[1]+  Running    sleep 100 &
```

Common Options

1. `jobs`

Displays all jobs.

```
jobs
```

Example:

```
[1]- Running    sleep 100 &
[2]+ Stopped    vim
```

---

2. `jobs -l`

Shows the **PID** along with job information.

```
jobs -l
```

Output:

```
[1]+ 13567 Running sleep 100 &
```

Here:

- Job ID = 1
- PID = 13567

---

3. `jobs -p`

Shows only the **process IDs**.

```
jobs -p
```

Output:

```
13567
```

4. `jobs -r`

Displays only **running** jobs.

```
jobs -r
```

Output:

```
[1]+ Running sleep 100 &
```

---

5. `jobs -s`

Displays only **stopped** jobs.

```
jobs -s
```

Output:

```
[2]+ Stopped vim
```

Job Symbols

Example:

```
jobs
```

Output:

```
[1]- Running sleep 100 &
[2]+ Stopped vim
```

Meaning:

Symbol	Meaning
+	Current (default) job
-	Previous job
No symbol	Older jobs

Using Job IDs

Bring a job to the foreground:

```
fg %1
```

Send a stopped job to the background:

```
bg %1
```

Terminate a job:

```
kill %1
```

Practical Example

```
sleep 500 &
sleep 600 &
jobs
```

Output:

```
[1]- Running    sleep 500 &
[2]+ Running    sleep 600 &
```

Bring job 1 to the foreground:

```
fg %1
```

Stop it:

```
Ctrl + Z
```

Resume it in the background:

```
bg %1
```

Check:

```
jobs
```

Output:

```
[1]- Running    sleep 500 &
[2]+ Running    sleep 600 &
```

Exit Status

- **0** – The command executed successfully.
- **Non-zero** – An error occurred (for example, an invalid option).

Notes

- `jobs` is a **Bash built-in command**, not an external executable.
- It only shows jobs started from the **current shell session**.
- It does **not** show all system processes. Use `ps` or `top` to view all processes.
- Background jobs started in one terminal are **not visible** in another terminal.

Summary

Command	Description
`jobs`	List all jobs in the current shell
`jobs -l`	List jobs with their PIDs
`jobs -p`	Display only process IDs
`jobs -r`	Show only running jobs
`jobs -s`	Show only stopped jobs
`fg %n`	Bring job *n* to the foreground
`bg %n`	Resume job *n* in the background
`kill %n`	Terminate job *n*

Example Workflow

```
sleep 300 &      # Start in background
jobs             # View jobs
fg %1            # Bring to foreground
Ctrl + Z         # Suspend it
bg %1            # Resume in background
jobs -l          # View job with PID
kill %1          # Terminate the job
jobs             # Confirm it is gone
```

This workflow demonstrates the most common use of `jobs` together with `bg`, `fg`, and `kill` for managing processes from the current shell.

11. kill

The `kill` command in Linux is used to **send a signal to a process**. Although its name suggests that it only terminates processes, it can actually send many different signals to control process behavior.

Syntax

```
kill [options] PID
```

- `PID` – Process ID of the process you want to signal.

Example:

```
kill 1234
```

Sends the default signal (`SIGTERM`) to process `1234`.

Common Signals

Signal	Number	Description
`SIGTERM`	15	Gracefully terminate a process (default).
`SIGKILL`	9	Forcefully kill a process. Cannot be ignored.
`SIGSTOP`	19	Pause (stop) a process.
`SIGCONT`	18	Resume a stopped process.
`SIGHUP`	1	Reload configuration or restart a daemon.
`SIGINT`	2	Interrupt a process (same as pressing **Ctrl+C**).

Examples

1. Gracefully terminate a process

```
kill 2456
```

or

```
kill -15 2456
```

or

```
kill -SIGTERM 2456
```

Attempts to let the program exit cleanly.

---

2. Forcefully kill a process

```
kill -9 2456
```

or

```
kill -SIGKILL 2456
```

Immediately stops the process.

Use this only if the process does not respond to `SIGTERM`.

---

3. Pause a process

```
kill -STOP 2456
```

The process is suspended.

---

4. Resume the paused process

```
kill -CONT 2456
```

Continues execution.

5. Reload a service

Many server applications reload their configuration with `SIGHUP`.

```
kill -HUP 2456
```

Find the Process ID

Using `ps`:

```
ps -ef
```

or

```
ps aux
```

Example:

```
bharath  2456  0.2  firefox
```

Kill it:

```
kill 2456
```

Using `pgrep`

Find a PID by process name:

```
pgrep firefox
```

Output:

```
2456
```

Kill it:

```
kill 2456
```

Kill Multiple Processes

```
kill 1234 5678 9012
```

List Available Signals

```
kill -l
```

Example output:

```
1) SIGHUP
2) SIGINT
3) SIGQUIT
9) SIGKILL
15) SIGTERM
18) SIGCONT
19) SIGSTOP
```

Check Whether the Process Was Killed

```
ps -p 2456
```

If nothing is displayed, the process has ended.

Exit Status

Exit Code	Meaning
`0`	Signal sent successfully.
Non-zero	An error occurred (e.g., invalid PID or insufficient permissions).

Important Notes

- `kill` sends **signals**; it does **not always terminate** a process.
- The default signal is **`SIGTERM` (15)**, which allows the process to clean up before exiting.
- **`SIGKILL` (9)** forcefully terminates the process and should be used only when a process does not respond to `SIGTERM`.
- You can only signal processes that you own unless you have superuser (`root`) privileges.

Quick Reference

Command	Purpose
`kill PID`	Gracefully terminate a process (`SIGTERM`).
`kill -9 PID`	Forcefully kill a process (`SIGKILL`).
`kill -15 PID`	Send `SIGTERM`.
`kill -STOP PID`	Pause a process.
`kill -CONT PID`	Resume a paused process.
`kill -HUP PID`	Reload configuration (for many daemons).
`kill -l`	List all available signals.

Example Session

```
$ sleep 300 &
[1] 4567

$ ps -p 4567
  PID TTY          TIME CMD
 4567 pts/0    00:00:00 sleep

$ kill 4567

$ ps -p 4567
# No output (the process has terminated)
```

This example starts a background `sleep` process, finds its PID (`4567`), and then terminates it using the default `SIGTERM` signal.

12. logout

The `logout` command is a shell built-in used to **end the current login shell session**. It logs the user out of the system if they are using a login shell.

Syntax

```
logout
```

or

```
logout [exit_status]
```

- `exit_status` (optional): Returns the specified exit status to the parent process.

What `logout` Does

- Terminates the current **login shell**.
- Closes the user's terminal session.
- Ends remote sessions (such as SSH) if it is the login shell.
- Returns control to the login prompt or disconnects the session.

Examples

1. Log out of the current session

```
logout
```

Output:

```
logout
Connection to server closed.
```

(if using SSH)

2. Logout with an exit status

```
logout 0
```

- Logs out successfully.
- Exit status is `0` (success).

---

3. Logout with an error status

```
logout 1
```

- Logs out and returns exit status `1`.

Example (SSH Session)

```
$ ssh user@192.168.1.10
user@server:~$ pwd
/home/user

user@server:~$ logout
Connection to 192.168.1.10 closed.
```

The SSH session ends.

Login Shell vs Non-Login Shell

`logout` only works in a **login shell**.

Example:

```
$ bash
$ logout
bash: logout: not login shell: use `exit`
```

Here, the new `bash` shell is **not** a login shell, so `logout` cannot be used.

Instead:

```
exit
```

or press

```
Ctrl + D
```

Difference Between `logout` and `exit`

Feature	`logout`	`exit`
Ends current shell	Yes	Yes
Works in login shell	Yes	Yes
Works in non-login shell	No	Yes
Used for logging out	Yes	Yes (if login shell)
Shell builtin	Yes	Yes

Exit Status

```
logout
echo $?
```

Typical values:

- `0` → Successful logout
- Non-zero → Indicates an error (if applicable)

Common Use Cases

- Logging out of an SSH session.
- Ending a terminal login session.
- Closing a virtual console (TTY) session.
- Exiting a login shell after completing work.

Notes

- In most modern shells (such as Bash), `logout` is a **shell built-in**, not an external executable.
- If there are **stopped jobs** running, the shell may prevent logout and display a warning:

```
logout
There are stopped jobs.
```

You can:

- Resume or terminate the jobs, then run `logout` again.
- Run `logout` a second time to force the logout (behavior depends on the shell).

Summary

- **Purpose:** End the current **login shell** and log the user out.
- **Syntax:** `logout [exit_status]`
- **Works only in:** Login shells.
- **Alternative:** Use `exit` (or `Ctrl + D`) for non-login shells.

13. pwd

The **pwd** command stands for **Print Working Directory**.

It displays the **full (absolute) path** of the directory you are currently working in.

Syntax

```
pwd
```

Example

```
$ pwd
/home/bharath
```

This means your current working directory is:

```
/home/bharath
```

If you change directories:

```
$ cd /etc
$ pwd
/etc
```

Now your current directory is `/etc`.

Why is pwd useful?

- To know your current location in the filesystem.
- Before creating, copying, moving, or deleting files.
- In shell scripts to determine the current directory.
- While navigating complex directory structures.

Common Examples

Example 1: Show current directory

```
$ pwd
/home/user/Documents
```

Example 2: After changing directory

```
$ cd /var/log
$ pwd
/var/log
```

Example 3: Use with another command

```
$ echo "Current directory is: $(pwd)"
```

Output:

```
Current directory is: /home/user
```

Example 4: Store the current directory in a variable

```
CURRENT_DIR=$(pwd)

echo $CURRENT_DIR
```

Output:

```
/home/user
```

Options

`-L` (Logical)

Displays the logical current directory, preserving symbolic links.

```
pwd -L
```

This is the **default** behavior in most shells.

Example:

```
/home/user/projects
```

Even if `projects` is a symbolic link, `pwd -L` shows the path through the symlink.

`-P` (Physical)

Displays the actual physical directory by resolving symbolic links.

```
pwd -P
```

Example:

Suppose:

```
/home/user/projects -> /mnt/storage/projects
```

Then:

```
pwd -L
```

Output:

```
/home/user/projects
```

```
pwd -P
```

Output:

```
/mnt/storage/projects
```

Difference between `pwd` and `echo $PWD`

Command	Description
`pwd`	Prints the current working directory by executing the `pwd` command (or shell builtin).
`echo $PWD`	Displays the value of the shell's `PWD` environment variable.

Normally, both produce the same output.

```
$ pwd
/home/user

$ echo $PWD
/home/user
```

Exit Status

- **0** – Command executed successfully.
- **Non-zero** – An error occurred (rare, such as if the current directory has become inaccessible).

Practical Example

```
$ pwd
/home/bharath

$ cd Downloads

$ pwd
/home/bharath/Downloads

$ mkdir test

$ cd test

$ pwd
/home/bharath/Downloads/test
```

This sequence shows how `pwd` reflects your current location after each `cd`.

Summary

Feature	Description
Command	`pwd`
Full Form	Print Working Directory
Purpose	Shows the current directory's absolute path
Common Use	Verify your location before performing file operations
Options	`-L` (logical path), `-P` (physical path)

`pwd` is one of the most frequently used Linux commands because it helps you keep track of where you are in the filesystem while navigating directories.

14. read

The `read` command is a **Bash built-in command** used to **accept input from the user** (or from a file/pipe) and store it in one or more variables.

Syntax

```
read [options] variable1 variable2 ...
```

If no variable is specified, the input is stored in the default variable `REPLY`.

Basic Example

```
read name
echo "Hello, $name"
```

Output:

```
John
Hello, John
```

The command waits for the user to type something and press **Enter**.

Prompting the User

Instead of using `echo`, you can display the prompt directly.

```
read -p "Enter your name: " name
echo "Welcome $name"
```

Output:

```
Enter your name: John
Welcome John
```

Reading Multiple Variables

```
read first last
```

Input:

```
John Smith
```

Variables:

```
first = John
last = Smith
```

Reading into the Default Variable (REPLY)

```
read
echo "$REPLY"
```

Input:

```
Linux
```

Output:

```
Linux
```

Reading a Password (Hidden Input)

```
read -s -p "Password: " pass
echo
echo "Password saved."
```

`-s` hides the characters as the user types.

Limiting the Number of Characters

```
read -n 1 answer
```

Input:

```
Y
```

The command stops after reading **one character**.

Setting a Timeout

```
read -t 5 name
```

The user has **5 seconds** to enter input.

If no input is received, `read` exits with a non-zero status.

Reading with a Different Delimiter

Normally, `read` stops at a newline.

```
read -d ":" value
```

Input:

```
hello:world
```

Output:

```
value=hello
```

Reading an Array

```
read -a fruits
```

Input:

```
apple banana mango
```

Access elements:

```
echo "${fruits[0]}"
echo "${fruits[1]}"
echo "${fruits[2]}"
```

Output:

```
apple
banana
mango
```

Reading a File Line by Line

```
while read line
do
    echo "$line"
done < file.txt
```

If `file.txt` contains:

```
Linux
Ubuntu
Fedora
```

Output:

```
Linux
Ubuntu
Fedora
```

Common Options

Option	Description
`-p`	Display a prompt before reading input
`-s`	Silent mode (hide typed characters)
`-n N`	Read exactly `N` characters
`-N N`	Read exactly `N` characters, even if Enter isn't pressed until all are read
`-t SEC`	Timeout after specified seconds
`-a ARRAY`	Store words into an array
`-d CHAR`	Stop reading at the specified delimiter
`-r`	Do not treat backslash (\) as an escape character
`-u FD`	Read from the specified file descriptor

Practical Example: Login Script

```
#!/bin/bash

read -p "Username: " user
read -s -p "Password: " pass
echo

echo "Welcome, $user!"
```

Example run:

```
Username: admin
Password:
Welcome, admin!
```

Practical Example: Menu

```
echo "1. Add User"
echo "2. Delete User"
echo "3. Exit"

read -p "Choose an option: " choice

case $choice in
    1) echo "Adding user..." ;;
    2) echo "Deleting user..." ;;
    3) echo "Exiting..." ;;
    *) echo "Invalid option" ;;
esac
```

Exit Status

- `0` – Input was read successfully.
- Non-zero – An error occurred, end-of-file (EOF) was reached, or a timeout occurred.

Example:

```
if read name; then
    echo "Hello $name"
else
    echo "No input received"
fi
```

Summary

- `read` accepts input from the keyboard, a file, or a pipe.
- It stores the input into one or more variables.
- It is commonly used in interactive shell scripts.
- Useful options include `-p` (prompt), `-s` (silent/password), `-t` (timeout), `-n`/`-N` (character limit), `-a` (array), and `-r` (raw input without interpreting backslashes).

15. set

The `set` command is a **Bash shell built-in** used to **display or change the behavior of the current shell**. It can also be used to **list all shell variables and functions**.

Syntax

```
set [options] [arguments]
```

or

```
set [+|-option]
```

- `-` enables an option.
- `+` disables an option.

Common Uses

1. Display all shell variables and functions

Running `set` without any options displays:

- Shell variables
- Environment variables
- Shell functions

```
set
```

Example (partial output):

```
BASH=/usr/bin/bash
HOME=/home/bharath
HOSTNAME=ubuntu
PATH=/usr/local/bin:/usr/bin
PWD=/home/bharath
USER=bharath
```

2. Enable shell debugging (`-x`)

Shows every command before it executes.

```
set -x
```

Example

```
#!/bin/bash
set -x

a=10
b=20
echo $((a+b))
```

Output

```
+ a=10
+ b=20
+ echo 30
30
```

Disable it:

```
set +x
```

---

3. Exit immediately if a command fails (`-e`)

Makes the script stop when a command returns a non-zero exit status.

Example

```
set -e

mkdir test
cd test

false

echo "This will not execute"
```

Output

```
mkdir: created directory 'test'
```

The script exits after `false`.

4. Treat undefined variables as errors (`-u`)

Example

```
set -u
echo $name
```

Output

```
bash: name: unbound variable
```

Without `set -u`, Bash would print an empty line.

---

5. Combine useful options

Many production scripts begin with:

```
set -euo pipefail
```

Meaning:

- `-e` → Exit on error
- `-u` → Undefined variables are errors
- `pipefail` → A pipeline fails if any command in it fails

Example

```
set -euo pipefail
```

This helps make scripts safer and easier to debug.

---

6. Show command execution and line numbers

```
set -vx
```

- `-v` prints each input line as read.
- `-x` prints each command after expansion, before execution.

Useful for debugging.

7. Disable filename expansion (globbing)

Normally:

```
echo *
```

lists all files.

Disable globbing:

```
set -f
```

Now:

```
echo *
```

Output

```
*
```

Enable again:

```
set +f
```

---

8. Set positional parameters

You can assign values to `$1`, `$2`, `$3`, etc.

```
set apple banana orange
```

Now:

```
echo $1
```

Output

```
apple
```

```
echo $2
```

Output

```
banana
```

```
echo $3
```

Output

```
orange
```

---

9. Check currently enabled shell options

```
set -o
```

Example output:

```
allexport       off
braceexpand     on
emacs           on
errexit         off
monitor         on
nounset         off
verbose         off
xtrace          off
```

---

10. Enable options using long names

Instead of:

```
set -e
```

you can write:

```
set -o errexit
```

Instead of:

```
set -u
```

use:

```
set -o nounset
```

Instead of:

```
set -x
```

use:

```
set -o xtrace
```

Frequently Used Options

Option	Long Name	Description
`-e`	`errexit`	Exit script if a command fails
`-u`	`nounset`	Error on undefined variables
`-x`	`xtrace`	Print commands before execution
`-v`	`verbose`	Print input lines as read
`-f`	`noglob`	Disable wildcard expansion (`*`, `?`)
`-a`	`allexport`	Automatically export new variables
`-o pipefail`	`pipefail`	Pipeline fails if any command fails
`+x`	—	Disable command tracing
`+e`	—	Disable exit-on-error
`+u`	—	Disable undefined-variable checking

Practical Example

```
#!/bin/bash

set -euo pipefail

name="Bharath"

echo "Hello $name"

mkdir demo

cd demo

touch file1

ls

false

echo "This line will never execute."
```

Output

```
Hello Bharath
file1
```

The script stops after `false` because `set -e` is enabled.

Summary

- `set` is a Bash built-in for viewing and changing shell behavior.
- Running `set` alone lists all shell variables and functions.
- `set -e` exits on command errors.
- `set -u` reports undefined variables.
- `set -x` traces commands as they execute.
- `set -o` displays shell options, and `set -o <name>` enables named options.
- `set` can also assign positional parameters (e.g., `set one two three` sets `$1`, `$2`, `$3`).
- A common best practice for robust Bash scripts is:

```
set -euo pipefail
```

This combination helps catch errors early and makes shell scripts more reliable.

16. source

The `source` command is a Bash built-in command that executes commands from a file **in the current shell** instead of starting a new shell.

Syntax

```
source filename
```

or the shorthand:

```
. filename
```

Both commands are equivalent.

Why use `source`?

Normally, when you execute a shell script:

```
./script.sh
```

or

```
bash script.sh
```

the script runs in a **new (child) shell**. Any variables or directory changes disappear when the script finishes.

When you use:

```
source script.sh
```

the commands run in the **current shell**, so all changes remain after the script completes.

Example 1: Setting Variables

Create a file named `test.sh`:

```
#!/bin/bash

NAME="Bharath"
```

Run normally:

```
bash test.sh
echo $NAME
```

Output:

The variable is not available.

Now use:

```
source test.sh
echo $NAME
```

Output:

```
Bharath
```

The variable remains because it was created in the current shell.

Example 2: Changing Directory

Create `change.sh`:

```
cd /etc
```

Run normally:

```
bash change.sh
pwd
```

Output:

```
/home/user
```

The directory doesn't change.

Now:

```
source change.sh
pwd
```

Output:

```
/etc
```

The current shell's directory changed.

Example 3: Reload `.bashrc`

A very common use of `source`:

```
source ~/.bashrc
```

or

```
. ~/.bashrc
```

Purpose:

- Reload aliases
- Reload environment variables
- Reload shell functions
- Apply changes without logging out

Example 4: Loading Environment Variables

Create `env.sh`:

```
export JAVA_HOME=/usr/lib/jvm/java-21-openjdk
export PATH=$JAVA_HOME/bin:$PATH
```

Load it:

```
source env.sh
```

Verify:

```
echo $JAVA_HOME
```

Output:

```
/usr/lib/jvm/java-21-openjdk
```

Example 5: Defining Functions

`functions.sh`

```
hello() {
    echo "Welcome!"
}
```

Load:

```
source functions.sh
```

Now you can call:

```
hello
```

Output:

```
Welcome!
```

Difference Between `bash` and `source`

Command	Runs in Current Shell?	Changes Persist?
`bash script.sh`	No	No
`./script.sh`	No	No
`source script.sh`	Yes	Yes
`. script.sh`	Yes	Yes

Common Uses

- Reload `.bashrc`
- Reload `.profile`
- Load environment variables
- Define aliases
- Define shell functions
- Change shell variables
- Update the `PATH` variable
- Configure development environments

Notes

- `source` is a **Bash built-in**, not an external executable.
- The file being sourced **does not need execute permission**, because Bash reads and executes it directly.

Example:

```
chmod -x myscript.sh
source myscript.sh
```

This still works.

Exit Status

The `source` command returns:

- `0` – Success
- Non-zero – An error occurred (for example, the file doesn't exist or contains an error)

Example:

```
source missing.sh
echo $?
```

Output:

```
bash: missing.sh: No such file or directory
1
```

Summary

- **Purpose:** Execute commands from a file in the **current shell**.
- **Syntax:**

  ```
  source filename
  ```

  or

  ```
  . filename
  ```

- **Main advantage:** Variables, functions, aliases, directory changes, and environment variables remain available after the file finishes executing.
- **Most common use:** Reload your shell configuration without opening a new terminal:

  ```
  source ~/.bashrc
  ```

17. type

The `type` command is a **Bash built-in command** that tells you **how the shell interprets a command name**. It helps you determine whether a command is:

- A shell built-in
- An alias
- A shell function
- An external executable (binary)
- A shell keyword

Syntax

```
type [OPTIONS] command_name
```

Basic Examples

1. Check if a command is built-in

```
type cd
```

Output:

```
cd is a shell builtin
```

Explanation:

- `cd` is built into Bash.
- There is no `/bin/cd` executable.

2. Check an external command

```
type ls
```

Output:

```
ls is /usr/bin/ls
```

Explanation:

- `ls` is an executable located in `/usr/bin/ls`.

---

3. Check an alias

Suppose you create:

```
alias ll='ls -l'
```

Now run:

```
type ll
```

Output:

```
ll is aliased to `ls -l'
```

---

4. Check a shell function

```
hello() {
    echo "Hello"
}
```

Now:

```
type hello
```

Output:

```
hello is a function
hello ()
{
    echo "Hello"
}
```

5. Check a shell keyword

```
type if
```

Output:

```
if is a shell keyword
```

Common Options

`-a`

Shows **all** matching commands found in the search order.

Example:

```
type -a ls
```

Output:

```
ls is aliased to `ls --color=auto'
ls is /usr/bin/ls
ls is /bin/ls
```

Useful when multiple versions exist.

`-p`

Prints only the path of external commands.

Example:

```
type -p ls
```

Output:

```
/usr/bin/ls
```

If the command is a built-in:

```
type -p cd
```

Output:

```
# No output
```

because `cd` has no executable file.

-P

Ignores aliases and functions and searches only the executable in PATH.

Example:

```
alias ls='ls --color=auto'
```

```
type ls
```

Output:

```
ls is aliased to `ls --color=auto'
```

Now:

```
type -P ls
```

Output:

```
/usr/bin/ls
```

-t

Prints only the type.

Example:

```
type -t cd
```

Output:

```
builtin
```

Other possible outputs:

```
alias
builtin
file
function
keyword
```

Example:

```
type -t ls
```

Output:

```
file
```

`-f`

Ignores shell functions.

Example:

```
type -f hello
```

If `hello` is only a function:

```
bash: type: hello: not found
```

Search Order Used by Bash

When you type a command, Bash searches in this order:

1. Alias
2. Shell keyword
3. Shell function
4. Shell built-in
5. Executable files in directories listed in `$PATH`

The `type` command shows which one Bash will use.

Difference Between `type`, `which`, and `command -v`

Command	Finds Aliases	Finds Built-ins	Finds Functions	Finds Executables
`type`				
`which`	(usually)			
`command -v`				

`type` is generally the most informative command for understanding how Bash resolves a command name.

Practical Examples

```
type echo
```

Output:

```
echo is a shell builtin
```

```
type pwd
```

Output:

```
pwd is a shell builtin
```

```
type grep
```

Output:

```
grep is /usr/bin/grep
```

```
type -t grep
```

Output:

```
file
```

```
type -a python3
```

Possible output:

```
python3 is /usr/local/bin/python3
python3 is /usr/bin/python3
```

When to Use ‘`type`’

- Determine whether a command is a built-in, alias, function, keyword, or external executable.
- Find the executable path for a command.
- Troubleshoot unexpected command behavior caused by aliases or functions.
- See all available versions of a command in your `PATH`.

For Bash users, `type` is one of the best tools for understanding exactly what will be executed when you enter a command.

18. ulimit

The `ulimit` command is a **Bash built-in command** used to **view or set limits on the system resources** that a shell and the processes started from it can use.

These limits help prevent a single process from consuming too many system resources.

Syntax

```
ulimit [options] [limit]
```

- Without a limit value, it displays the current limit.
- With a limit value, it sets the specified limit.

Types of Limits

There are two kinds of limits:

- **Soft limit** – Current operating limit. Users can increase it up to the hard limit.
- **Hard limit** – Maximum limit. Only the root user can increase it.

Display soft limit:

```
ulimit -S
```

Display hard limit:

```
ulimit -H
```

Common Options

Option	Resource Controlled
`-a`	Display all limits
`-c`	Maximum core dump size
`-d`	Maximum data segment size
`-f`	Maximum file size
`-l`	Maximum locked memory
`-m`	Maximum physical memory (obsolete on many systems)
`-n`	Maximum number of open files
`-p`	Pipe buffer size
`-s`	Stack size
`-t`	CPU time limit
`-u`	Maximum number of user processes
`-v`	Maximum virtual memory
`-x`	Maximum file locks (on some systems)

Examples

1. Display all limits

```
ulimit -a
```

Example output:

```
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max user processes              (-u) 30922
open files                      (-n) 1024
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
virtual memory          (kbytes, -v) unlimited
```

2. Check maximum open files

```
ulimit -n
```

Output:

```
1024
```

This means the current shell can have **1024 open file descriptors**.

---

3. Increase open files

```
ulimit -n 4096
```

Now verify:

```
ulimit -n
```

Output:

```
4096
```

This change only affects the current shell session (subject to the hard limit).

---

4. View stack size

```
ulimit -s
```

Output:

```
8192
```

Meaning:

```
8192 KB (8 MB)
```

5. Set stack size

```
ulimit -s 16384
```

Sets the stack size to **16 MB**.

---

6. View CPU time limit

```
ulimit -t
```

Output:

```
unlimited
```

---

7. Limit CPU time

```
ulimit -t 30
```

A process started from this shell is limited to **30 seconds of CPU time**. If it exceeds this, the operating system sends a signal to terminate it.

---

8. Check maximum user processes

```
ulimit -u
```

Output:

```
30922
```

---

9. Limit user processes

```
ulimit -u 100
```

The shell can create at most **100 processes**.

10. File size limit

```
ulimit -f
```

Set a limit:

```
ulimit -f 1000
```

Limits newly created files to **1000 blocks** (the exact block size depends on the system).

---

11. Core dump size

View:

```
ulimit -c
```

Enable unlimited core dumps:

```
ulimit -c unlimited
```

Disable core dumps:

```
ulimit -c 0
```

Soft vs Hard Limit Example

View the soft limit:

```
ulimit -Sn
```

View the hard limit:

```
ulimit -Hn
```

Example output:

```
Soft : 1024
Hard : 4096
```

You can increase the soft limit up to the hard limit:

```
ulimit -Sn 4096
```

Trying to exceed the hard limit as a normal user will result in an error.

Make Limits Permanent

The `ulimit` command changes limits only for the current shell session. To make changes persistent, configure PAM limits in:

```
/etc/security/limits.conf
```

or add files under:

```
/etc/security/limits.d/
```

Example:

```
bharath soft nofile 4096
bharath hard nofile 8192
```

After logging in again, the new limits will apply (assuming PAM is configured to use `pam_limits`).

Real-World Uses

- Limit the number of files a web server can open.
- Prevent applications from consuming excessive memory.
- Restrict CPU usage for long-running jobs.
- Limit the number of processes a user can create.
- Enable or disable core dump generation for debugging.
- Tune resource limits for database and application servers.

Summary

Command	Description
`ulimit -a`	Show all resource limits
`ulimit -n`	Show maximum open files
`ulimit -n 4096`	Set maximum open files
`ulimit -s`	Show stack size
`ulimit -s 16384`	Set stack size
`ulimit -u`	Show maximum user processes
`ulimit -u 100`	Set maximum user processes
`ulimit -c`	Show core dump size
`ulimit -c unlimited`	Enable unlimited core dumps
`ulimit -f`	Show maximum file size
`ulimit -t`	Show CPU time limit
`ulimit -Sn`	Show soft limit for open files
`ulimit -Hn`	Show hard limit for open files

Note: `ulimit` is a **shell built-in**, not an external executable. Limits set with `ulimit` affect only the current shell and any child processes it starts unless they are configured system-wide.

19. umask

The `umask` command is used to **set or display the default permissions** assigned to newly created files and directories.

A **umask** does **not** assign permissions directly. Instead, it **removes (masks)** permissions from the default permissions.

Syntax

```
umask
```

Display the current umask value.

```
umask MASK
```

Set a new umask value.

```
umask -S
```

Display the mask in symbolic format.

Default Permissions

When Linux creates a new object, it starts with these maximum permissions:

Object	Default Permission	Symbolic
File	666	rw-rw-rw-
Directory	777	rwxrwxrwx

Files are **not** created with execute (`x`) permission by default.

The umask subtracts permissions from these defaults.

Permission Calculation

Formula

For files:

```
666 - umask
```

For directories:

```
777 - umask
```

Example 1

Current umask:

```
umask
```

Output

```
0022
```

New file:

```
666 - 022 = 644
```

Permission becomes

```
-rw-r--r--
```

New directory:

```
777 - 022 = 755
```

Permission becomes

```
drwxr-xr-x
```

Example 2

Set

```
umask 077
```

Create a file

```
touch secret.txt
```

Permissions

```
-rw-------
```

Create a directory

```
mkdir private
```

Permissions

```
drwx------
```

Only the owner can access them.

Example 3

Set

```
umask 002
```

Create

```
touch project.txt
mkdir project
```

Permissions

File

```
-rw-rw-r--
```

Directory

```
drwxrwxr-x
```

Useful when users in the same group need write access.

View Current umask

```
umask
```

Example

```
0022
```

View in Symbolic Form

```
umask -S
```

Example

```
u=rwx,g=rx,o=rx
```

This means:

- User: read, write, execute
- Group: read, execute
- Others: read, execute

Change umask Temporarily

```
umask 027
```

Check

```
umask
```

Output

```
0027
```

This remains in effect only for the current shell session.

Make umask Permanent

For a single user, add the command to:

`~/.bashrc`

Example:

```
umask 027
```

Reload:

```
source ~/.bashrc
```

Or log out and back in.

For all users, administrators may configure a default umask in system-wide shell initialization files (such as `/etc/profile`) or via PAM/login configuration, depending on the Linux distribution.

Common umask Values

umask	File Permission	Directory Permission	Description
000	666 (rw-rw-rw-)	777 (rwxrwxrwx)	No permissions removed (least secure)
022	644 (rw-r--r--)	755 (rwxr-xr-x)	Common default on many Linux systems
027	640 (rw-r-----)	750 (rwxr-x---)	Group can read, others have no access
077	600 (rw-------)	700 (rwx------)	Private files and directories
002	664 (rw-rw-r--)	775 (rwxrwxr-x)	Good for collaborative group work

---

Practical Example

```
$ umask
0022

$ touch file1

$ ls -l file1
-rw-r--r-- 1 user user 0 Jul 19 06:30 file1
```

Change the umask:

```
$ umask 077

$ touch file2

$ ls -l file2
-rw------- 1 user user 0 Jul 19 06:31 file2
```

Common Uses

- Protect private files (`umask 077`)
- Allow group collaboration (`umask 002`)
- Set organization-wide default permissions
- Improve security by restricting access to newly created files

---

Summary

Command	Description
`umask`	Display the current umask value
`umask 022`	Set the umask to `022`
`umask 077`	Create private files/directories by default
`umask 002`	Allow group members to write new files
`umask -S`	Show the umask in symbolic form

Key point: `umask` does **not** grant permissions. It only **removes permissions** from the default creation permissions (`666` for files and `777` for directories).

20. unalias

The **`unalias`** command is used to **remove an alias** that was previously created with the `alias` command.

Syntax

```
unalias alias_name
```

or

```
unalias -a
```

Purpose

Aliases are shortcuts for longer commands. The `unalias` command deletes those shortcuts.

For example, if you have:

```
alias ll='ls -l'
```

Typing:

```
ll
```

is equivalent to:

```
ls -l
```

To remove this shortcut:

```
unalias ll
```

Now:

```
ll
```

produces:

```
bash: ll: command not found
```

(unless `ll` is defined elsewhere).

Options

Option	Description
`-a`	Remove all aliases

Example:

```
unalias -a
```

This deletes every alias in the current shell session.

Examples

Example 1: Create and remove an alias

Create an alias:

```
alias c='clear'
```

Check it:

```
alias
```

Output:

```
alias c='clear'
```

Remove it:

```
unalias c
```

Verify:

```
alias
```

The alias `c` is no longer listed.

Example 2: Remove multiple aliases

Create aliases:

```
alias l='ls'
alias h='history'
alias cls='clear'
```

Remove two of them:

```
unalias l h
```

Remaining:

```
alias
```

Output:

```
alias cls='clear'
```

Example 3: Remove all aliases

View aliases:

```
alias
```

Remove all:

```
unalias -a
```

Verify:

```
alias
```

Output:

```
(no aliases displayed)
```

Check if an Alias Exists

Create an alias:

```
alias greet='echo Hello'
```

Check:

```
alias greet
```

Output:

```
alias greet='echo Hello'
```

Remove it:

```
unalias greet
```

Check again:

```
alias greet
```

Output:

```
bash: alias: greet: not found
```

Common Error

If you try to remove an alias that doesn't exist:

```
unalias abc
```

Output:

```
bash: unalias: abc: not found
```

To suppress the error:

```
unalias abc 2>/dev/null
```

Temporary vs Permanent

`unalias` only affects the **current shell session**.

If an alias is defined in files such as:

```
~/.bashrc
~/.bash_aliases
/etc/bash.bashrc
```

it will be recreated the next time you open a terminal or source those files.

To remove it permanently:

1. Open `~/.bashrc` (or `~/.bash_aliases`).
2. Delete or comment out the alias definition.
3. Reload the configuration:

```
source ~/.bashrc
```

Summary

Command	Description
`unalias ll`	Remove the alias `ll`
`unalias c`	Remove the alias `c`
`unalias -a`	Remove all aliases
`alias`	List all current aliases
`alias name`	Show a specific alias

Key Points

- `unalias` removes aliases created with `alias`.
- It is a **Bash built-in command**.
- `unalias alias_name` removes one alias.
- `unalias -a` removes all aliases.
- Changes apply only to the current shell unless you also remove the alias from your shell startup files.

21. unset

The `unset` command is a **Bash built-in command** used to **remove variables or shell functions** from the current shell environment.

Syntax

```
unset [OPTION] NAME
```

- `NAME` – Variable or function to remove.

---

1. Unsetting a Variable

When you unset a variable, it no longer exists in the current shell.

Example

```
name="Bharath"
echo $name
```

Output:

```
Bharath
```

Now remove it:

```
unset name
```

Check again:

```
echo $name
```

Output:

The variable has been deleted.

2. Unsetting Multiple Variables

```
a=10
b=20
c=30

unset a b c
```

Verify:

```
echo $a $b $c
```

Output:

---

3. Unsetting an Environment Variable

Exported variables can also be removed.

```
export PATH1="/home/user/bin"

echo $PATH1
```

Output:

```
/home/user/bin
```

Remove it:

```
unset PATH1
```

---

4. Unsetting a Shell Function

Create a function:

```
hello() {
    echo "Hello World"
}
```

Run it:

```
hello
```

Output:

```
Hello World
```

Delete the function:

```
unset -f hello
```

Now:

```
hello
```

Output:

```
bash: hello: command not found
```

---

5. Using `-v` Option

`-v` explicitly tells Bash to unset a variable.

```
x=100
unset -v x
```

Equivalent to:

```
unset x
```

Options

Option	Description
`-v`	Remove a shell variable (default behavior)
`-f`	Remove a shell function

Practical Examples

Example 1: Remove a Temporary Variable

```
temp="/tmp/files"

echo $temp
unset temp
echo $temp
```

Example 2: Remove a Function

```
backup() {
    echo "Backing up files..."
}

backup

unset -f backup

backup
```

Output:

```
Backing up files...
bash: backup: command not found
```

Example 3: Clear a Secret Variable

```
password="MySecret123"

unset password
```

Now:

```
echo $password
```

Output:

Things to Remember

- `unset` affects **only the current shell session**.
- It **does not delete files**—only variables or functions.
- Once a variable is unset, its value is lost unless you assign it again.
- Some special shell variables are **read-only** and cannot be unset.

Example:

```
unset UID
```

Output:

```
bash: unset: UID: cannot unset: readonly variable
```

Difference Between `unset` and Setting an Empty Value

```
name="Bharath"
```

```
name=""
```

The variable still exists:

```
echo "$name"
```

Output:

Check if it exists:

```
declare -p name
```

Output:

```
declare -- name=""
```

Using `unset`:

```
unset name
```

Now:

```
declare -p name
```

Output:

```
bash: declare: name: not found
```

Difference:

Command	Result
`name=""`	Variable exists but contains an empty string
`unset name`	Variable is completely removed

Summary

Command	Purpose
`unset var`	Delete a variable
`unset -v var`	Delete a variable explicitly
`unset -f func`	Delete a function
`unset a b c`	Delete multiple variables
`unset PATH1`	Remove an exported environment variable from the current shell

`unset` is commonly used in shell scripts and interactive Bash sessions to clean up temporary variables, remove functions, or clear environment variables when they are no longer needed.