BASH

PROGRAMMING

BASH

PROGRAMMING LANGUAGE

SCRIPTING THE GNU/ LINUX SHELL

Bash Scripting Guide

Introduction

About This Guide

This guide aims to aid people interested in learning to work with BASH. It aspires to teach good practice techniques for using BASH and writing simple scripts.

This guide is targeted at beginning users. It assumes no advanced knowledge – just the ability to log in to a Unix-like system and open a command-line (terminal) interface. It will help if you know how to use a text editor; we will not be covering editors, nor do we endorse any particular editor choice. Familiarity with the fundamental Unix toolset or with other programming languages or programming concepts is not required, but those who have such knowledge may understand some of the examples more quickly.

If something is unclear to you, you are invited to report this so that it may be clarified in this document for future readers.

You are invited to contribute to the development of this document by extending it or correcting invalid or incomplete information.

A Definition

BASH is an acronym for **B**ourne **A**gain **Sh**ell. It is based on the Bourne shell and is mostly compatible with its features.

Shells are command interpreters. They are applications that provide users with the ability to give commands to their operating system interactively or to execute batches of commands quickly. In no way are they required for the execution of programs; they are merely a layer between system function calls and the user.

Think of a shell as a way for you to speak to your system. Your system doesn’t need it for most of its work, but it is an excellent interface between you and what your system can offer. It allows you to perform basic math, run basic tests, and execute applications. More importantly, it allows you to combine these operations and connect applications to each other to perform complex and automated tasks.

BASH is not your operating system. It is not your window manager. It is not your terminal (but it often runs inside your terminal). It does not control your mouse or keyboard. It does not configure your system, activate your screensaver, or open your files when you double-click them. It is generally not involved in launching applications from your window manager or desktop environment. It’s important to understand that BASH is only an interface for you to execute statements (using BASH syntax), either at the interactive BASH prompt or via BASH scripts.

- **Shell**: A (possibly interactive) command interpreter, acting as a layer between the user and the system.
- **Bash**: The Bourne Again Shell, a Bourne-compatible shell.

Using Bash

Most users think of BASH as a prompt and a command line. That is BASH in interactive mode. BASH can also run in non-interactive mode, as when executing scripts. We can use scripts to automate certain logic. Scripts are basically lists of commands (just like the ones you can type on the command line), but stored in a file. When a script is executed, all these commands are (generally) executed sequentially, one after another.

We'll start with the basics in an interactive shell. Once you're familiar with those, you can put them together in scripts.

Important! You should make yourself familiar with the `man` and `apropos` commands on the shell. They will be vital to your self-tutoring.

```
 $ man man
$ man apropos
```

In this guide, the `$` at the beginning of a line represents your BASH prompt. Traditionally, a shell prompt either ends with `$`, `%`, or `#`. If it ends with `$`, this indicates a shell that's compatible with the Bourne shell (such as a POSIX shell, Korn shell, or Bash). If it ends with `%`, this indicates a C shell (csh or tcsh); this guide does not cover C shell. If it ends with `#`, this indicates that the shell is running as the system's superuser account (`root`), and you should be extra careful.

Your actual BASH prompt will probably be much longer than `$`. Prompts are often highly individualized.

The `man` command stands for "manual"; it opens documentation (so-called "man pages") on various topics. You use it by running the command `man [topic]` at the BASH prompt, where `[topic]` is the name of the "page" you wish to read. Note that many of these "pages" are considerably longer than one printed page; nevertheless, the name persists. Each command (application) on your system is likely to have a man page. There are pages for other things too, such as system calls or specific configuration files. In this guide, we will only be covering commands.

Commands and Arguments

BASH reads commands from its input (which is usually either a terminal or a file). Each line of input that it reads is treated as a command — an instruction to be carried out. (There are a few advanced cases, such as commands that span multiple lines, that will be covered later.)

Bash divides each line into words that are demarcated by a whitespace character (spaces and tabs). The first word of the line is the name of the command to be executed. All the remaining words become arguments to that command (options, filenames, etc.).

Assume we're in an empty directory... (to try these commands, create an empty directory called `test` and enter that directory by running: `mkdir test; cd test`):

```
 $ ls                # List files in the current directory (no output, no files).
$ touch a b c       # Create files 'a', 'b', and 'c'.
$ ls                # List files again, and this time outputs: 'a', 'b', and 'c'.
a  b  c
```

The command `ls` prints out the names of the files in the current directory. The first time we run the list command, we get no output because there are no files yet.

The `#` character at the start of a word indicates a comment. Any words following the comment are ignored by the shell, meant only for reading. If we run these examples in our own shell, we don't have to type the comments; but even if we do, the command will still work.

`touch` is an application that changes the Last Modified time of a file. If the filename that it is given does not exist yet, it creates a file of that name as a new and empty file. In this example, we passed three arguments. `touch` creates a file for each argument. `ls` shows us that three files have been created.

```
         $ rm *             # Remove all files in the current directory.
        $ ls                # List files in the current directory (no output, no files).
        $ touch a    b c    # Create files 'a', 'b', and 'c'.
        $ ls                # List files again; this time the outputs: 'a', 'b', and 'c'.
        a   b   c
```

`rm` is an application that removes all the files that it was given. `*` is a glob that basically means "all" and in this case means all files in the current directory. We will talk more about globs later.

Now, did we notice that there are several spaces between `a` and `b`, and only one between `b` and `c`? Also, notice that the files that were created by `touch` are no different than the first time. The amount of whitespace between arguments does not matter! This is important to know. For example:

```
         $ echo This is a test.
        This is a test.
        $ echo This    is    a    test.
        This is a test.
```

`echo` is a command that prints its arguments to standard output (which in our case is the terminal). In this example, we provide the `echo` command with four arguments: `This`, `is`, `a`, and `test.`. `echo` takes these arguments and prints them out one by one with a space in between. In the second case, the exact same thing happens. The extra spaces make no difference. If we want the extra whitespace, we need to pass the sentence as one single argument. We can do this by using quotes:

```
         $ echo "This    is    a    test."
        This    is    a    test.
```

Quotes group everything inside them into a single argument. The argument is: `This    is    a    test.` ... specifically spaced. Note that the quotes are not part of the argument — Bash removes them before handing the argument to `echo`. `echo` prints this single argument out just like it always does.

Be very careful to avoid the following:

```
         $ ls                                       # There are two files in the cur
directory.
        The secret voice in your head.mp3  secret
        $ rm The secret voice in your head.mp3        # Executes rm with 6 arguments; n
        rm: cannot remove `The': No such file or directory
        rm: cannot remove `voice': No such file or directory
        rm: cannot remove `in': No such file or directory
        rm: cannot remove `your': No such file or directory
        rm: cannot remove `head.mp3': No such file or directory
        $ ls                                          # List files in the current direc
It is still there.
        The secret voice in your head.mp3             # But your file 'secret' is now g
```

We need to make sure we quote filenames properly. If we don't, we'll end up deleting the wrong things!

`rm` takes filenames as arguments. If our filenames have spaces and we do not quote them, Bash thinks each word is a separate argument. Bash hands each argument to `rm` separately.

To fix this, we quote the filename:

```
 $ rm "The secret voice in your head.mp3"
$ ls
secret
```

Now `rm` receives one argument, the full filename, and deletes the correct file.

Special Characters

Some characters are evaluated by Bash to have a non-literal meaning. Instead, these characters carry out a special instruction or have an alternate meaning; they are called “special characters” or “meta-characters”.

Here are some of the more common special characters and their uses:

Char.	Description
" "	**Whitespace** — this is a tab, newline, vertical tab, form feed, carriage return, or space. Bash uses whitespace to determine where words begin and end. The first word is the command name, and additional words become arguments to that command.
`$`	**Expansion** — introduces various types of expansion: parameter expansion (e.g., `$var` or `${var}`), command substitution (e.g., `$(command)`), or arithmetic expansion (e.g., `$((expression))`). More on expansions later.
`'`	**Single quotes** — protect the text inside them so that it has a literal meaning. With them, generally any kind of interpretation by Bash is ignored: special characters are passed over, and multiple words are prevented from being split.
`"`	**Double quotes** — protect the text inside them from being split into multiple words or arguments, yet allow substitutions to occur; the meaning of most other special characters is usually prevented.
`\`	**Escape** — (backslash) prevents the next character from being interpreted as a special character. This works outside of quoting, inside double quotes, and is generally ignored in single quotes.
`#`	**Comment** — the `#` character begins a commentary that extends to the end of the line. Comments are notes of explanation and are not processed by the shell.
`=`	**Assignment** — assigns a value to a variable (e.g., `logdir=/var/log/myprog`). Whitespace is not allowed on either side of the `=` character.
`[[ ]]`	**Test** — an evaluation of a conditional expression to determine whether it is “true” or “false”. Tests are used in Bash to compare strings, check the existence of a file, etc. More on this later.

`!`	**Negate** — used to negate or reverse a test or exit status. For example: `! grep text file; exit $?` .
`>` , `>>` , `<`	**Redirection** — redirect a command’s output or input to a file. Redirections will be covered later.
`|`	**Pipe** — sends the output from one command to the input of another command. This is a method of chaining commands together. Example: `echo "Hello beautiful." | grep -o beautiful` .
`;`	**Command separator** — used to separate multiple commands that are on the same line.
`{ }`	**Inline group** — commands inside the curly braces are treated as if they were one command. It is convenient to use these when Bash syntax requires only one command, and a function doesn’t feel warranted.
`( )`	**Subshell group** — similar to the above but where commands within are executed in a subshell (a new process). Used much like a sandbox; if a command causes side effects (like changing variables), it will have no effect on the current shell.
`(( ))`	**Arithmetic expression** — with an arithmetic expression, characters such as `+` , `-` , `*` , and `/` are mathematical operators used for calculations. They can be used for variable assignments like `(( a = 1 + 4 ))` as well as tests like `if (( a < b ))` . More on this later.
`$(( ))`	**Arithmetic expansion** — comparable to the above, but the expression is replaced with the result of its arithmetic evaluation. Example: `echo "The average is $(( (a+b)/2 ))"` .
`*` , `?`	**Globs** — “wildcard” characters that match parts of filenames (e.g., `ls *.txt`).
`~`	**Home directory** — the tilde is a representation of a home directory. When alone or followed by a `/` , it means the current user’s home directory; otherwise, a username must be specified (e.g., `ls ~/Documents; cp ~john/.bashrc .`).
`&`	**Background** — when used at the end of a command, runs the command in the background (do not wait for it to complete).

Examples:

```
 $ echo "I am $LOGNAME"
I am lhunath
$ echo 'I am $LOGNAME'
I am $LOGNAME
$ # boo
$ echo An open\\ \  space
An open   space
$ echo "My computer is $(hostname)"
My computer is Lyndir
$ echo boo > file
$ echo $(( 5 + 5 ))
10
$ (( 5 > 0 )) && echo "Five is greater than zero."
```

```
Five is greater than zero.
```

Deprecated Special Characters (Recognized, but Not Recommended)

Some characters are evaluated by Bash to have a non-literal meaning but are not recommended for use:

- **Backticks ( ) for Command Substitution**: Use `$(...)` instead of backticks for command substitution. The `$(...)` syntax is clearer, nestable, and POSIX-compliant.

  Example:

  ```
   # Deprecated
  echo
  x60;date x60;x60
  # Recommended
  echo $(date)
  ```

Parameters

Parameters are a sort of named space in memory you can use to retrieve or store information. Generally speaking, they will store string data, but can also be used to store integers, indexed, and associative arrays.

Parameters come in two flavors: **variables** and **special parameters**. Special parameters are read-only, pre-set by BASH, and used to communicate some type of internal status. Variables are parameters that you can create and update yourself. Variable names are bound by the following rule:

- **Name**: A word consisting only of letters, digits, and underscores, and beginning with a letter or an underscore. Also referred to as an identifier.

To store data in a variable, we use the following assignment syntax:

```
 $ varname=vardata
```

This command assigns the data `vardata` to the variable by name of `varname` .

Please note that you cannot use spaces around the `=` sign in an assignment. If you write this:

```
 # This is wrong!
$ varname = vardata
```

Bash will not know that you are attempting to assign something. The parser will see `varname` with no `=` and treat it as a command name, then pass `=` and `vardata` to it as arguments.

To access the data stored in a variable, we use **parameter expansion**. Parameter expansion is the substitution of a parameter by its value, which is to say, the syntax tells Bash that you want to use the contents of the variable. After that, Bash may still perform additional manipulations on the result. This is a very important concept to grasp correctly, because it is very much unlike the way variables are handled in other programming languages!

Bash Parameter Expansion Cheat Sheet

	Simpler Description

Syntax	
`${parameter:-word}`	**Use a Default Value.** If the variable is empty or doesn’t exist, use `word` instead. The original variable isn’t changed.
`${parameter:=word}`	**Set a Default Value.** If the variable is empty or doesn’t exist, set its value to `word`. The variable is permanently changed.
`${parameter:+word}`	**Use an Alternate Value.** If the variable *does* exist, use `word`. If it’s empty or doesn’t exist, do nothing.
`${parameter:offset:length}`	**Get a Substring.** This lets you grab a piece of the variable’s value. Start `offset` characters in and take `length` characters.
`${#parameter}`	**Get the Length.** Tells you how many characters are in the variable’s value.
`${parameter#pattern}`	**Remove from Start (Shortest).** Deletes the shortest bit of text from the *beginning* of the variable that matches the `pattern`.
`${parameter##pattern}`	**Remove from Start (Longest).** Deletes the longest bit of text from the *beginning* of the variable that matches the `pattern`.
`${parameter%pattern}`	**Remove from End (Shortest).** Deletes the shortest bit of text from the *end* of the variable that matches the `pattern`.
`${parameter%%pattern}`	**Remove from End (Longest).** Deletes the longest bit of text from the *end* of the variable that matches the `pattern`.
`${parameter/pat/string}`	**Replace First Match.** Finds the first piece of text matching `pat` and replaces it with `string`.
`${parameter//pat/string}`	**Replace All Matches.** Finds every piece of text matching `pat` and replaces all of them with `string`.
`${parameter/#pat/string}`	**Replace at Start.** If the text at the *beginning* of the variable matches `pat`, it’s replaced with `string`.
`${parameter/%pat/string}`	**Replace at End.** If the text at the *end* of the variable matches `pat`, it’s replaced with `string`.

Usage Examples

Command	Example
`${parameter:-word}`	`name=""; echo "${name:- John}"`
`${parameter:=word}`	`name=""; echo "${name:=John}"; echo "$name"`
`${parameter:+word}`	`name="Jane"; echo "${name:+is set}"`

`${parameter:offset:length}`	`path="/usr/bin/bash"; echo "${path:5:3}"`
`${#parameter}`	`path="/usr/bin/bash"; echo "${#path}"`
`${parameter#pattern}`	`path="/usr/bin/bash"; echo "${path#*/}"`
`${parameter##pattern}`	`path="/usr/bin/bash"; echo "${path##*/}"`
`${parameter%pattern}`	`file="image.jpg.bak"; echo "${file%.*}"`
`${parameter%%pattern}`	`file="image.jpg.bak"; echo "${file%%.*}"`
`${parameter/pat/string}`	`path="/usr/bin/bash"; echo "${path/bin/sbin}"`
`${parameter//pat/string}`	`path="/usr/bin/bash/bash"; echo "${path//bash/sh}"`
`${parameter/#pat/string}`	`path="usr/bin/bash"; echo "${path/#/home/}"`
`${parameter/%pat/string}`	`file="image.jpg"; echo "${file/%.jpg/.png}"`

To illustrate what parameter expansion is, let's use this example:

```
 $ foo=bar
$ echo "Foo is $foo"
Foo is bar
```

When Bash is about to execute your code, it first changes the command by taking your parameter expansion (the `$foo`), and replacing it by the contents of `foo`, which is `bar`. The command becomes:

```
 $ echo "Foo is bar"
Foo is bar
```

Now, Bash is ready to execute the command. Executing it shows us the simple sentence on screen.

It is important to understand that parameter expansion causes the `$parameter` to be replaced by its contents. Note the following case, which relies on an understanding of argument splitting:

```
 $ song="My song.mp3"
$ rm $song
rm: My: No such file or directory
rm: song.mp3: No such file or directory
```

Why did this not work? Because Bash replaced your `$song` by its contents, being `My song.mp3`; then it performed word splitting; and only THEN executed the command. It was as if you had typed this:

```
 $ rm My song.mp3
```

And suddenly, according to the rules of word splitting, Bash thought you meant for `My` and `song.mp3` to mean two different files, because there is whitespace between them, and it wasn't quoted. How do we fix this? We remember to put double quotes around every parameter expansion!

```
 $ rm "$song"
```

- **Parameters**: Parameters store data that can be retrieved through a symbol or a name.

Special Parameters and Variables

Let’s get our vocabulary straight before we get into the real deal. There are **Parameters** and **Variables**. Variables are actually just one kind of parameter: parameters that are denoted by a name. Parameters that aren’t variables are called **Special Parameters**. I’m sure you’ll understand things better with a few examples:

```
 $ # Some parameters that aren't variables:
$ echo "My shell is $0, and has these options set: $-"
My shell is -bash, and has these options set: himB
$ # Some parameters that ARE variables:
$ echo "I am $LOGNAME, and I live at $HOME."
I am lhunath, and I live at /home/lhunath.
```

Please note: Unlike PHP/Perl/... parameters do NOT start with a `$` -sign. The `$` -sign you see in the examples merely causes the parameter that follows it to be expanded. Expansion basically means that the shell replaces the parameter by its content. As such, `LOGNAME` is the parameter (variable) that contains your username. `$LOGNAME` is an expression that will be replaced with the content of that variable, which in my case is `lhunath` .

Here’s a summary of most of the Special Parameters:

Parameter Name	**Usage**	**Description**
`0`	`"$0"`	Contains the name, or the path, of the script. This is not always reliable.
`1` , `2` , etc.	`"$1"` , `"$2"` , etc.	Positional Parameters contain the arguments that were passed to the current script or function.
`*`	`"$*"`	Expands to all the words of all the positional parameters. Double quoted, it expands to a single string containing them all, separated by the first character of the `IFS` variable.
`@`	`"$@"`	Expands to all the words of all the positional parameters. Double quoted, it expands to a list of them as individual words.
`#`	`$#`	Expands to the number of positional parameters that are currently set.
`?`	`$?`	Expands to the exit code of the most recently completed foreground command.
`-`	`$-`	Expands to the option flags that are currently set.
`$`	`$$`	Expands to the process ID (PID) of the Bash shell running the current script.
		Expands to the process ID of the most

`!`	`$!`	recent background command.
`_`	`$_`	Expands to the last argument of the last command executed.

Examples:

```
 $ # A script that demonstrates the use of special parameters
$ cat myscript.sh
#!/usr/bin/env bash
echo "Script name: $0"
echo "First argument: $1"
echo "All arguments: $@"
echo "Number of arguments: $#"
echo "Last command's exit status: $?"
$ ./myscript.sh foo bar
Script name: ./myscript.sh
First argument: foo
All arguments: foo bar
Number of arguments: 2
Last command's exit status: 0
```

Patterns

BASH offers three different kinds of pattern matching. Pattern matching serves two roles in the shell: selecting filenames within a directory or determining whether a string conforms to a desired format.

On the command line, you will mostly use **globs**. These are a fairly straightforward form of patterns that can easily be used to match a range of files or to check variables against simple rules.

The second type of pattern matching involves **extended globs**, which allow more complicated expressions than regular globs.

Since version 3.0, Bash also supports **regular expression patterns**. These will be useful mainly in scripts to test user input or parse data. (You can't use a regular expression to select filenames; only globs and extended globs can do that.)

- **Pattern**: A pattern is a string with a special format designed to match filenames or to check, classify, or validate data strings.

Glob Patterns

Globs are a very important concept in Bash, if only for their incredible convenience. Properly understanding globs will benefit you in many ways. Globs are basically patterns that can be used to match filenames or other strings.

Globs are composed of normal characters and metacharacters. Metacharacters are characters that have a special meaning. These are the metacharacters that can be used in globs:

- `*` : Matches any string, including the null string.
- `?` : Matches any single character.
- `[...]` : Matches any one of the enclosed characters.

Globs are implicitly anchored at both ends. What this means is that a glob must match a whole string (filename or data string). A glob of `a*` will not match the string `cat`, because it only matches the `at`, not the whole string. A glob of `ca*`, however, would match `cat`.

Here's an example of how we can use glob patterns to expand to filenames:

```
 $ ls
a  abc  b  c
$ echo *
 a abc b c
$ echo a*
 a abc
```

Bash sees the glob, for example `a*`. It expands this glob by looking in the current directory and matching it against all files there. Any filenames that match the glob are gathered up and sorted, and then the list of filenames is used in place of the glob. As a result, the statement `echo a*` is replaced by the statement `echo a abc`, which is then executed.

When a glob is used to match filenames, the `*` and `?` characters cannot match a slash (`/`) character. So, for instance, the glob `*/bin` might match `foo/bin` but it cannot match `/usr/local/bin`. When globs match patterns, the `/` restriction is removed.

Bash performs filename expansions **after** word splitting has already been done. Therefore, filenames generated by a glob will not be split; they will always be handled correctly. For example:

```
 $ touch "a b.txt"
$ ls
a b.txt
$ rm *
$ ls
```

Here, `*` is expanded into the single filename `a b.txt`. This filename will be passed as a single argument to `rm`. Using globs to enumerate files is always a better idea than using `ls` for that purpose. Here's an example with some more complex syntax, which we will cover later on, but it will illustrate the reason very well:

```
 $ ls
a b.txt
$ for file in `ls`; do rm "$file"; done
rm: cannot remove `a': No such file or directory
rm: cannot remove `b.txt': No such file or directory
$ for file in *; do rm "$file"; done
$ ls
```

Here, we use the `for` command to go through the output of the `ls` command. The `ls` command prints the string `a b.txt`. The `for` command splits that string into words over which it iterates. As a result, `for` iterates over first `a`, and then `b.txt`. Naturally, this is not what we want. The glob, however, expands in the proper form. It results in the string `a b.txt`, which `for` takes as a single argument.

In addition to filename expansion, globs may also be used to check whether data matches a specific format. For example, we might be given a filename and need to take different actions depending on its extension:

```
 $ filename="somefile.jpg"
```

```
$ if [[ $filename = *.jpg ]]; then
> echo "$filename is a jpeg"
> fi
somefile.jpg is a jpeg
```

The `[[` keyword and the `case` keyword (which we will discuss in more detail later) both offer the opportunity to check a string against a glob – either regular globs or extended globs, if the latter have been enabled.

> **Good Practice:**
>
> You should always use globs instead of `ls` (or similar) to enumerate files. Globs will always expand safely and minimize the risk for bugs. You can sometimes end up with some very weird filenames. Most scripts aren't tested against all the odd cases that they may end up being used against.

Extended Globs

Extended globs provide more complex pattern-matching capabilities. To use them, you must enable them with:

```
 shopt -s extglob
```

Here are the extended glob patterns:

- `*(pattern)` : Matches zero or more occurrences of the given pattern.
- `+(pattern)` : Matches one or more occurrences of the given pattern.
- `?(pattern)` : Matches zero or one occurrence of the given pattern.
- `@(pattern)` : Matches exactly one occurrence of the given pattern.
- `!(pattern)` : Matches anything except the given pattern.

Example:

```
 $ shopt -s extglob
$ ls
abc.txt  def.txt  ghi.jpg
$ echo *.@(txt|jpg)
abc.txt def.txt ghi.jpg
$ echo !(*.txt)
ghi.jpg
```

Extended globs are particularly useful in scripts for more precise filename matching or string validation.

Tests and Conditionals

Bash Conditionals & Arithmetic Constructs

In Bash you have several constructs for evaluation and testing: - `[ ]` → POSIX test command (strings, numbers, files) - `[[ ]]` → Bash/Ksh test command (extended, safer with strings/regex) - `(( ))` → arithmetic evaluation (integer math, conditional context) - `$(( ))` → arithmetic expansion (substitute result into commands)

Syntax	Type / Context	Purpose / Usage
`[ ]`	POSIX test command (external binary)	String comparison, integer tests (`-eq` , `-lt` , `-gt`), file checks (`-f` , `-d` , `-e` , etc).
`[[ ]]`	Bash/Ksh keyword	String tests (quotes not required), supports regex (`=~`), safer than `[ ]` with whitespace and globbing.
`(( ))`	Arithmetic evaluation command	C-style integer math. Assign variables, test numeric conditions. Exit status: 0 if expr !=0, 1 if expr == 0.
`$(( ))`	Arithmetic expansion	Expands to value of arithmetic expression. Use inside commands to capture numeric result.

Arithmetic Operators

Within an arithmetic expression, operators are evaluated in order of precedence. The following table lists the operators from highest to lowest precedence.

Operator	Description
`id++` `id--`	Variable post-increment and post-decrement
`++id` `--id`	Variable pre-increment and pre-decrement
`-` `+`	Unary minus and plus
`!` `~`	Logical and bitwise negation
`**`	Exponentiation
`*` `/` `%`	Multiplication, division, remainder
`+` `-`	Addition, subtraction
`<<` `>>`	Left and right bitwise shifts
`<=` `>=` `<` `>`	Comparison
`==` `!=`	Equality and inequality
`&`	Bitwise AND
`^`	Bitwise exclusive OR
`|`	Bitwise OR
`&&`	Logical AND
`||`	Logical OR
`expr ? expr : expr`	Conditional operator
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`	Assignment
`,`	Comma

Examples:

```
 # [ ] POSIX test
if [ "$x" = "hello" ]; then
  echo "x is hello"
fi

# [[ ]] extended test
if [[ "$y" =~ ^[0-9]+$ ]]; then
  echo "y contains only digits"
fi

# (( )) arithmetic evaluation
count=5
if (( count > 3 )); then
  echo "count is greater than 3"
fi
(( count++ ))  # increment

# $(( )) arithmetic expansion
echo "2+3 = $(( 2 + 3 ))"
z=$(( 10 * 4 ))
echo "z = $z"
```

Conclusion

Conceptually, you can think of these constructs as “playgrounds”: - `[ ]` → the original POSIX test playground - `[[ ]]` → safer and extended string playground - `(( ))` → arithmetic evaluation playground - `$(( ))` → arithmetic expansion playground

Sequential execution of commands is one thing, but to achieve any advanced logic in your scripts or your command-line one-liners, you’ll need tests and conditionals. Tests determine whether something is true or false. Conditionals are used to make decisions that determine the execution flow of a script.

1. Exit Status

Every command results in an exit code whenever it terminates. This exit code is used by whatever application started it to evaluate whether everything went OK. This exit code is like a return value from functions. It’s an integer between 0 and 255 (inclusive). Convention dictates that we use 0 to denote success, and any other number to denote failure of some sort. The specific number is entirely application-specific and is used to hint at what exactly went wrong.

For example, the `ping` command sends ICMP packets over the network to a certain host. That host normally responds to this packet by sending the exact same one right back. This way, we can check whether we can communicate with a remote host. `ping` has a range of exit codes that can tell us what went wrong, if anything did:

If `ping` does not receive any reply packets at all, it will exit with code 1. If a packet count and deadline are both specified, and fewer than count packets are received by the time the deadline has arrived, it will also exit with code 1. On other errors, it exits with code 2. Otherwise, it exits with code 0. This makes it possible to use the exit code to see if a host is alive or not.

The special parameter `?` shows us the exit code of the last foreground process that terminated. Let’s play around a little with `ping` to see its exit codes:

```
 $ ping Good
ping: unknown host Good
$ echo $?
2
$ ping -c 1 -W 1 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
--- 1.1.1.1 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
$ echo $?
1
```

Good Practice:

You should make sure that your scripts always return a non-zero exit code if something unexpected happened in their execution. You can do this with the `exit` builtin:

```
rm file || { echo 'Could not delete file!' >&2; exit 1; }
```

- **Exit Code / Exit Status**: Whenever a command ends, it notifies its parent (which in our case will always be the shell that started it) of its exit status. This is represented by a number ranging from 0 to 255. This code is a hint as to the success of the command's execution.

2. Control Operators (&& and ||)

Now that we know what exit codes are, and that an exit code of '0' means the command's execution was successful, we'll learn to use this information. The easiest way of performing a certain action depending on the success of a previous command is through the use of control operators. These operators are `&&` and `||`, which respectively represent a logical AND and a logical OR. These operators are used between two commands, and they are used to control whether the second command should be executed depending on the success of the first. This concept is called conditional execution.

Let's put that theory into practice:

```
 $ mkdir d && cd d
```

This simple example has two commands, `mkdir d` and `cd d`. You could use a semicolon to separate the commands and execute them sequentially; but we want something more. In the above example, Bash will execute `mkdir d`, then `&&` will check the result of the `mkdir` application after it finishes. If the `mkdir` application was successful (exit code 0), then Bash will execute the next command, `cd d`. If `mkdir d` failed and returned a non-0 exit code, Bash will skip the next command, and we will stay in the current directory.

Another example:

```
 $ rm /etc/some file.conf || echo "I couldn't remove the file"
rm: cannot remove `/etc/some_file.conf': No such file or directory
I couldn't remove the file
```

`||` is much like `&&`, but it does the exact opposite. It only executes the next command if the first failed. As such, the message is only echoed if the `rm` command was unsuccessful.

In general, it's not a good idea to string together multiple different control operators in one command.

`&&` and `||` are quite useful in simple cases, but not in complex ones. In the next few sections, we'll show some other tools you can use for decision-making.

> **Good Practice:**
>
> It's best not to get overzealous when dealing with conditional operators. They can make your script hard to understand, especially for a person that's assigned to maintain it later.

Understanding Tests (`[` vs `[[`)

Bash provides two ways to evaluate conditional expressions: the `test` command (for which `[` is an alias) and the more modern `[[ ... ]]` keyword. While they share many operators, `[[` is generally safer, more powerful, and preferred for Bash scripts.

Here is a breakdown of the tests they support.

Tests Supported by Both `[` and `[[`

- **File Tests**

Operator	Description
`-e FILE`	True if file exists.
`-f FILE`	True if file is a regular file.
`-d FILE`	True if file is a directory.
`-h FILE`	True if file is a symbolic link.
`-p PIPE`	True if pipe exists.
`-r FILE`	True if file is readable by you.
`-s FILE`	True if file exists and is not empty.
`-t FD`	True if file descriptor `FD` is opened on a terminal.
`-w FILE`	True if the file is writable by you.
`-x FILE`	True if the file is executable by you.
`-O FILE`	True if the file is effectively owned by you.
`-G FILE`	True if the file is effectively owned by your group.
`FILE1 -nt FILE2`	True if `FILE1` is newer than `FILE2` .
`FILE1 -ot FILE2`	True if `FILE1` is older than `FILE2` .

- **String Tests**

Operator	Description
`-z STRING`	True if the string is empty.
`-n STRING`	True if the string is not empty.

`STRING = STRING`	True if the strings are identical.
`STRING != STRING`	True if the strings are not identical.
`STRING < STRING`	True if the first string sorts before the second (in current locale).
`STRING > STRING`	True if the first string sorts after the second (in current locale).

- **Numeric Tests**

Operator	Description
`INT1 -eq INT2`	True if the integers are equal.
`INT1 -ne INT2`	True if the integers are not equal.
`INT1 -lt INT2`	True if the first integer is less than the second.
`INT1 -gt INT2`	True if the first integer is greater than the second.
`INT1 -le INT2`	True if the first integer is less than or equal to the second.
`INT1 -ge INT2`	True if the first integer is greater than or equal to the second.

Additional Features of `[[`

The `[[` keyword supports all the above tests plus several powerful additions:

Operator	Description
`STRING == PATTERN`	True if the string matches the glob pattern.
`STRING != PATTERN`	True if the string does not match the glob pattern.
`STRING =~ REGEX`	True if the string matches the regular expression.
`( EXPR )`	Parentheses for grouping expressions.
`EXPR && EXPR`	Logical AND with short-circuit evaluation.
`EXPR || EXPR`	Logical OR with short-circuit evaluation.

Features Exclusive to `[` (and `test`)

These are generally discouraged in favor of using multiple `[` commands or switching to `[[`.

Operator	Description
`EXPR -a EXPR`	Logical AND.
`EXPR -o EXPR`	Logical OR.

Examples

```
          # Check if a file exists
```

```
$ test -e /etc/hosts && echo "Hosts file found."
Hosts file found.

# Check if a variable is set
$ test -n "$HOME" && echo "Your home directory is set."
Your home directory is set.

# String comparison with [[
$ [[ boar != bear ]] && echo "Boars aren't bears."
Boars aren't bears.

# Glob pattern matching with [[
$ [[ "boar" != "b?ar" ]] && echo "This will not print."

# Regex matching with [[
$ [[ "abc-123" =~ ^[a-z]+-[0-9]+$ ]] && echo "Matches regex."
Matches regex.
```

Good Practice:

- When writing **Bash** scripts, you should always prefer `[[` over `[`. It prevents common errors with word splitting and quoting and provides more features like pattern matching.
- When writing portable **shell** scripts that might run in environments without Bash, you must use `[` as it is more portable.
- Avoid the `-a` and `-o` operators within `[`. They are obsolete and their behavior can be unpredictable. Use multiple `[` commands instead: `bash     if [ "$food" = "apple" ] && [ "$drink" = "tea" ]; then       echo "The meal is acceptable."     fi`

Loops

Loops in Bash let you repeat blocks of code. The main loop types are `for`, `while`, and `until`.

1. For Loops

(a) Iterate over a list of values

```
 for color in red green blue; do
    echo "Color: $color"
done
```

(b) Iterate over command output

A safer way to iterate over command output, especially filenames, is to use globs.

```
 for file in *.sh; do
    # Skips the loop if no .sh files are found
    [ -e "$file" ] || continue
    echo "Script: $file"
done
```

(c) C-style syntax

```
for ((i=0; i<5; i++)); do
    echo "Number: $i"
done
```

(d) With ranges

```
for i in {1..5}; do
    echo "Range: $i"
done
```

2. While Loops

Loops while a condition is true.

```
count=1
while [ $count -le 5 ]; do
    echo "Count is $count"
    ((count++))
done
```

You can also read a file line by line:

```
while IFS= read -r line; do
    echo "Line: $line"
done < /etc/passwd
```

3. Until Loops

The opposite of `while`; it loops UNTIL a condition is true.

```
count=1
until [ $count -gt 5 ]; do
    echo "Until loop: $count"
    ((count++))
done
```

4. Loop Control

- `break [N]`: Exits out of loops.
  - Without `N`, it breaks only the innermost loop.
  - With `N`, it breaks out of `N` levels of loops.
- `continue [N]`: Skips to the next iteration of the loop.
  - Without `N`, it applies to the innermost loop.
  - With `N`, it applies to the `N`th enclosing loop.

```
# Break example
for i in {1..5}; do
    if [ $i -eq 3 ]; then
        echo "Breaking at $i"
        break
    fi
    echo "i=$i"
done

# Continue example
for i in {1..5}; do
    if [ $i -eq 3 ]; then
        echo "Skipping $i"
        continue
    fi
    echo "i=$i"
done
```

Arrays

As mentioned earlier, Bash has multiple parameter types. Of importance here, variables can either contain a single value (scalars) or multiple values (arrays).

Strings are without a doubt the most used parameter type. But they are also the most misused parameter type. It is important to remember that a string holds just one element. Capturing the output of a command, for instance, and putting it in a string parameter means that parameter holds just one string of characters, regardless of whether that string represents twenty filenames, twenty numbers, or twenty names of people.

And as is always the case when you put multiple items in a single string, these multiple items must be somehow delimited from each other. We, as humans, can usually decipher what the different filenames are when looking at a string. We assume that, perhaps, each line in the string represents a filename, or each word represents a filename. While this assumption is understandable, it is also inherently flawed. Each single filename can contain every character you might want to use to separate the filenames from each other in a string. That means there's technically no telling where the first filename in the string ends, because there's no character that can say: "I denote the end of this filename" because that character itself could be part of the filename.

Often, people make this mistake:

```
 # This does NOT work in the general case
$ files=$(ls ~/*.jpg); cp $files /backups/
```

When this would probably be a better idea (using array notation, which is explained later in this section):

```
 # This DOES work in the general case
$ files=(~/*.jpg); cp "${files[@]}" /backups/
```

The first attempt at backing up our files in the current directory is flawed. We put the output of `ls` in a string called `files` and then use the unquoted `$files` parameter expansion to cut that string into arguments (relying on word splitting). As mentioned before, argument and word splitting cuts a string into pieces wherever there is whitespace. Relying on it means we assume that none of our filenames will contain any whitespace. If they do, the filename will be cut in half or more. Conclusion: bad.

The only safe way to represent multiple string elements in Bash is through the use of arrays. An array is a type of variable that maps integers to strings. That basically means that it holds a numbered list of strings. Since each of these strings is a separate entity (element), it can safely contain any character, even whitespace.

For the best results and the least headaches, remember that if you have a list of things, you should always put it in an array.

Unlike some other programming languages, Bash does not offer lists, tuples, etc. Just arrays and associative arrays (which are new in Bash 4).

- **Array**: An array is a numbered list of strings: It maps integers to strings.

Creating Arrays

There are several ways you can create or fill your array with data. There is no one single true way: the method you'll need depends on where your data comes from and what it is.

The easiest way to create a simple array with data is by using the `=()` syntax:

```
 $ names=("Bob" "Peter" "$USER" "Big Bad John")
```

This syntax is great for creating arrays with static data or a known set of string parameters, but it gives us very little flexibility for adding lots of array elements. If you need more flexibility, you can also specify explicit indexes:

```
 $ names=([0]="Bob" [1]="Peter" [20]="$USER" [21]="Big Bad John")
# or...
$ names[0]="Bob"
```

Notice that there is a gap between indices 1 and 20 in this example. An array with holes in it is called a **sparse array**. Bash allows this, and it can often be quite useful.

If you want to fill an array with filenames, then you'll probably want to use globs in there:

```
 $ photos=(~"/My Photos"/*.jpg)
```

Notice here that we quoted the `My Photos` part because it contains a space. If we hadn't quoted it, Bash would have split it up into `photos=('~/My' 'Photos/'*.jpg )`, which is obviously not what we want. Also notice that we quoted only the part that contained the space. That's because we cannot quote the `~` or the `*`; if we do, they'll become literal, and Bash won't treat them as special characters anymore.

Unfortunately, it's really easy to incorrectly create arrays with a bunch of filenames in the following way:

```
 $ files=$(ls)    # BAD, BAD, BAD!
$ files=($(ls))  # STILL BAD!
```

Remember to always avoid using `ls`. The first would create a string with the output of `ls`. That string cannot possibly be used safely for reasons mentioned in the Arrays introduction. The second is closer, but it still splits the output of `ls` into words, which can break on filenames with spaces.

To correctly capture filenames, use a glob:

```
 $ files=( * )  # Correct: captures all files in the current directory
```

Accessing Array Elements

To access array elements, use the `${array[index]}` syntax:

```
$ echo "${names[0]}"  # Prints "Bob"
$ echo "${names[@]}"  # Prints all elements as separate words
$ echo "${names[*]}"  # Prints all elements as a single string
$ echo "${!names[@]}" # Prints all indices (0 1 20 21)
$ echo "${#names[@]}" # Prints the number of elements (4)
```

When quoted, `"${array[@]}"` and `"${array[*]}"` have a crucial difference. `"${array[@]}"` expands each element of the array into a separate word, preserving spaces within each element. This is almost always what you want. In contrast, `"${array[*]}"` expands into a single word, with all array elements joined by the first character of the `IFS` (Internal Field Separator) variable, which is a space by default.

The `@` symbol refers to all elements, and `!` gives the indices. Always quote array expansions to preserve whitespace in elements:

```
$ for file in "${files[@]}"; do echo "Processing $file"; done
```

Good Practice:

Always use arrays for lists of items, especially filenames. Quote array expansions to prevent word splitting. Avoid parsing `ls` output.

Associative Arrays

Since Bash 4.0, you can use associative arrays, which use strings as keys instead of integers. This is similar to dictionaries or maps in other programming languages.

To create an associative array, you must first declare it using `declare -A`.

```
# Declare an associative array
declare -A user

# Add elements
user[name]="John Doe"
user[email]="john.doe@example.com"
user[city]="New York"
```

You can also initialize it at declaration:

```
declare -A colors=( [red]="#FF0000" [green]="#00FF00" [blue]="#0000FF" )
```

Accessing Associative Array Elements

Access elements using the key inside the brackets. Remember to quote the expansion.

```
echo "User's name is: ${user[name]}"
echo "The hex code for red is: ${colors[red]}"
```

To list all keys:

```
echo "All keys: ${!user[@]}"
```

To list all values:

```
 echo "All values: ${user[@]}"
```

Iterating Over Associative Arrays

You can loop over the keys to get each key-value pair.

```
 for key in "${!user[@]}"; do
    echo "Key: $key, Value: ${user[$key]}"
done
```

> **Good Practice:**
>
> Associative arrays are very useful for storing related data. Always declare them with `declare -A` before use.

Simulating Multidimensional Arrays

Why Simulate Multidimensional Arrays? Bash (Bourne Again SHell) natively supports only one-dimensional arrays (both indexed and associative). This limitation arises because Bash is primarily designed for simple scripting tasks like file manipulation and process control, not complex data structures. For tasks requiring multidimensional data (e.g., matrices, grids, or tables like a 2D spreadsheet), we must simulate them to:

- Handle structured data efficiently, such as game boards, configuration matrices, or data processing without external tools like awk or Python.
- Avoid performance overhead from calling external programs.
- Maintain portability in pure Bash environments where multidimensional support isn't available (unlike languages like Python or C++).

Simulation methods include flattening into 1D arrays with index calculations, using associative arrays with composite keys, or nesting arrays via naming conventions. These workarounds can be error-prone (e.g., off-by-one errors in indexing) but are effective for small-scale use.

Example 1: Flattening a 2D Array into a 1D Indexed Array with Index Calculation

This simulates a 3x3 matrix by treating it as a 1D array of 9 elements. Access elements via row * cols + col.

```
 #!/bin/bash

# Define dimensions
rows=3
cols=3

# Declare a 1D array
declare -a matrix

# Initialize the matrix (e.g., fill with values 1 to 9)
for ((i=0; i<rows*cols; i++)); do
    matrix[$i]=$((i+1))
done
```

```
# Function to get value at (row, col)
get_value() {
    local row=$1 col=$2
    echo "${matrix[$((row * cols + col))]}"
}

# Function to set value at (row, col)
set_value() {
    local row=$1 col=$2 value=$3
    matrix[$((row * cols + col))]=$value
}

# Usage
echo "Initial matrix:"
for ((r=0; r<rows; r++)); do
    for ((c=0; c<cols; c++)); do
        printf "%d " "$(get_value $r $c)"
    done
    echo
done

set_value 1 1 99  # Set center to 99

echo "Modified matrix:"
for ((r=0; r<rows; r++)); do
    for ((c=0; c<cols; c++)); do
        printf "%d " "$(get_value $r $c)"
    done
    echo
done
```

Example 2: Using Associative Arrays with Composite Keys

This uses Bash 4+ associative arrays where keys are strings like “row,col”. Ideal for sparse or non-numeric data.

```
 #!/bin/bash

# Declare associative array
declare -A matrix

# Initialize some values
matrix[0,0]="A" matrix[0,1]="B" matrix[0,2]="C"
matrix[1,0]="D" matrix[1,1]="E" matrix[1,2]="F"
matrix[2,0]="G" matrix[2,1]="H" matrix[2,2]="I"

# Function to get value
get_value() {
    local key="$1,$2"
    echo "${matrix[$key]}"
}

# Function to set value
set_value() {
    local key="$1,$2"
    matrix[$key]=$3
}

# Usage: Print matrix
echo "Initial matrix:"
for r in {0..2}; do
    for c in {0..2}; do
        printf "%s " "$(get_value $r $c)"
    done
    echo
done

set_value 1 1 "X"  # Change center to X

echo "Modified matrix:"
for r in {0..2}; do
    for c in {0..2}; do
        printf "%s " "$(get_value $r $c)"
    done
    echo
done
```

Example 3: Nesting Arrays via Naming Conventions (Array of Arrays)

This simulates 2D by creating separate 1D arrays for each row, prefixed with a name like “rowX”.

```
 #!/bin/bash

# Define rows
rows=3

# Initialize row arrays
declare -a row0=(10 20 30)
declare -a row1=(40 50 60)
declare -a row2=(70 80 90)

# Function to get value
get_value() {
    local row=$1 col=$2
    local array_name="row${row}[${col}]"
    echo "${!array_name}"
}

# Function to set value
set_value() {
    local row=$1 col=$2 value=$3
    local array_name="row${row}"
    eval "${array_name}[${col}]=${value}"
}

# Usage: Print matrix
echo "Initial matrix:"
for ((r=0; r<rows; r++)); do
    for ((c=0; c<3; c++)); do  # Assuming fixed columns=3
        printf "%d " "$(get_value $r $c)"
    done
    echo
```

```
        done

        set_value 1 1 99  # Set row1 col1 to 99

        echo "Modified matrix:"
        for ((r=0; r<rows; r++)); do
            for ((c=0; c<3; c++)); do
                printf "%d " "$(get_value $r $c)"
            done
            echo
        done
```

These examples are self-contained and can be copied into separate scripts for testing. For larger datasets, consider switching to a language with native support like Python.

Input and Output

Input and output in Bash scripts is a complex topic because there is a great deal of flexibility in how it's done. This chapter will only scratch the surface of what is possible.

Input refers to any information that your program receives (or reads). Input to a Bash script can come from several different places:

- Command-line arguments (which are placed in the positional parameters)
- Environment variables, inherited from whatever process started the script
- Files
- Anything else a File Descriptor can point to (pipes, terminals, sockets, etc.). This will be discussed below.

Output refers to any information that your program produces (or writes). Output from a Bash script can also go to lots of different places:

- Files
- Anything else a File Descriptor can point to
- Command-line arguments to some other program
- Environment variables passed to some other program

Input and output are important in shell script programming. Figuring out where your input comes from, what it looks like, and what you must do to it to produce your desired output are core requirements for almost all scripts.

1. Command-line Arguments

For many scripts, the first (or the only) input we will care about are the arguments received by the script on the command line. As we saw in the Parameters chapter, there are some Special Parameters available to every script that contain these arguments. These are called the **Positional Parameters**. They are a very simple numerically indexed array of strings (in fact, in the POSIX shell, they are the only array the shell has). The first positional parameter is referred to with `$1`; the second, with `$2`; and so on. After the 9th one, you must use curly braces to refer to them: `${10}`, `${11}`, etc. But in practice, it's exceedingly rare that you would ever need to do that, because there are better ways to deal with them as a group.

In addition to referring to them one at a time, you may also refer to the entire set of positional parameters with the `"$@"` substitution. The double quotes here are extremely important. If you don't use the double quotes, each one of the positional parameters will undergo word splitting and globbing. You don't want that. By using the quotes, you tell Bash that you want to preserve each parameter as a separate word.

Another way to deal with the positional parameters is to eliminate each one as it is used. There is a special builtin command named `shift` that is used for this purpose. When you issue the `shift` command, the first positional parameter (`$1`) goes away. The second one becomes `$1` , the third one becomes `$2` , and so on down the line. So, if you wish, you can write a loop that keeps using `$1` over and over.

In real scripts, a combination of these techniques is used. A loop to process `$1` as long as it begins with a `-` takes care of the options. Then, when all the options have been processed and shifted away, everything that's left (in `"$@"`) is presumably a filename that we want to process.

Example:

```
         while [[ $1 == -* ]]; do
            case $1 in
                -v) verbose=1 ;;        -h) echo "Usage: $0 [-v] [-h] files..."; exit 0
*) echo "Unknown option: $1" >&2; exit 1 ;;    esac    shift
        done
        for file in "$@"; do
            echo "Processing $file"
        done
```

Good Practice:

Identify where your input comes from before you start writing. If you get to design the data flow into your script, then choose a way that makes sense for the kind of data you're dealing with. If you need to pass filenames, passing them as arguments is an excellent approach, because each one is encapsulated as a word, ready to go.

2. The Environment

Every program inherits certain information, resources, privileges, and restrictions from its parent process. One of those resources is a set of variables called **Environment Variables**.

In Bash, environment variables work very much like the regular shell variables we're used to. The only real difference is that they're already set when the script starts up; we don't have to set them ourselves.

Traditionally, environment variables have names that are all capital letters, such as `PATH` or `HOME` . This helps you avoid creating any variables that would conflict with them; as long as your variables all contain at least one lowercase letter, you should never have to worry about accidentally colliding with the environment. (Bash's special variables are also capitalized, such as `PIPESTATUS` . This is done for the exact same reason – so you can avoid having your variables trampled by Bash.)

Passing information to a program through the environment is useful in many situations. One of those is user preference. Not every user on a Unix-like system has the same likes and dislikes in applications, and in some cases, they may not all speak the same language. So, it's useful for users to be able to tell every application they run what their favorite editor is (the `EDITOR` environment variable) or what language they speak (the various environment variables that compose the user's locale). Environment variables can be set in each user's dotfiles, and then they will be passed automatically to every application the user runs from the login session.

Example:

```
 $ echo "My home directory is $HOME"
My home directory is /home/lhunath
$ export MY_VAR="Hello"
$ bash -c 'echo $MY_VAR'
Hello
```

Here, `export` makes `MY_VAR` an environment variable, so it's passed to the new `bash` process.

3. File Descriptors and Redirection

File descriptors (FDs) are the way Unix systems handle input and output. The three standard file descriptors are:

- **0 (stdin)**: Standard input, typically the keyboard or a redirected file.
- **1 (stdout)**: Standard output, typically the terminal.
- **2 (stderr)**: Standard error, also typically the terminal.

You can redirect these using `>`, `>>`, and `<`:

- `command > file`: Redirects stdout to `file`, overwriting it.
- `command >> file`: Redirects stdout to `file`, appending to it.
- `command < file`: Redirects stdin from `file`.
- `command 2> file`: Redirects stderr to `file`.

Example:

```
 $ echo "Hello" > output.txt
$ cat < output.txt
Hello
$ ls /nonexistent 2> error.log
$ cat error.log
ls: cannot access
'/nonexistent': No such file or directory
```

You can also redirect one FD to another:

```
 $ ls /nonexistent > output.txt 2>&1  # Redirects stderr to stdout, then both t
output.txt
```

Pipes (`|`) connect the stdout of one command to the stdin of another:

```
 $ echo "Hello world" | grep world
Hello world
```

4. Heredocs and Herestrings

Sometimes, storing multi-line data in a separate file is overkill. Heredocs and herestrings allow you to embed data directly inside your script, redirecting it to the standard input of a command.

a. Here Documents (Heredocs)

A Here Document, or **heredoc**, is a way to pass multiple lines of input to a command. The syntax uses a redirection operator `<<` followed by a delimiter. The block of text ends when the same delimiter is found on a line by itself.

```
$ grep proud <<END
> I am a proud sentence.
> And a second line.
> END
I am a proud sentence.
```

The delimiter can be any word (e.g., `END` , `EOF` , `STOP`). The key is that the closing delimiter must be on a new line, by itself, with no leading or trailing spaces.

Substitution

By default, the shell performs parameter expansion and command substitution within a heredoc.

```
$ cat <<EOF
> My user is: $USER
> My current directory is: $(pwd)
> EOF
My user is: osimage
My current directory is: /home/osimage/Downloads
```

To disable substitutions and treat the text literally, quote the delimiter:

```
$ cat <<'EOF'
> My user is: $USER
> My current directory is: $(pwd)
> EOF
My user is: $USER
My current directory is: $(pwd)
```

Indentation

To improve script readability, you might want to indent your heredoc content. Using `<<-` instead of `<<` will cause Bash to strip all leading **tab** characters (not spaces) from the input and the delimiter line.

```
if [[ some condition ]]; then
    # Note the use of tabs for indentation
    cat <<-EOF
    This line is indented.
    So is this one.
    The shell will remove the leading tabs.
    EOF
fi
```

b. Here Strings (Herestrings)

A **herestring** (`<<<`) is a more compact version of a heredoc. It's used to pass a single string (often from a variable) to a command's standard input.

```
$ wc -w <<< "This is a short sentence."
```

```
5
```

This is more efficient and often clearer than the alternative `echo "$VAR" | command`, as it doesn't create a separate subshell for the `echo` command.

```
 $ my_var="Some data to process"
# Preferred method:
$ read -r -a words <<< "$my_var"

# Clumsier alternative:
$ read -r -a words < <(echo "$my_var")
```

c. Advanced Usage with `cat` and Heredocs

A very common pattern is using `cat` with a heredoc to write multi-line content to a file.

Writing a Heredoc to a File

```
 # Overwrite file.txt with the content
cat <<EOF > file.txt
Line 1
Line 2 with a $VARIABLE
EOF

# Append the content to file.txt
cat <<EOF >> file.txt
Line 3
Line 4
EOF
```

Nesting Heredocs

You can nest heredocs, which is useful in complex scripts, for example when generating a script that itself contains a heredoc. The key is to use different delimiters for the inner and outer blocks.

```
 # This example runs a command on a remote server that
# creates a script file (/tmp/remote_script.sh) on that server.
ssh user@remote_host <<'OUTER_EOF'
  echo "Creating a script on the remote host..."

  # The inner heredoc writes the script content.
  # Note the different delimiter (INNER_EOF).
  cat <<'INNER_EOF' > /tmp/remote_script.sh
#!/bin/bash
# This is the script that will be created on the remote machine.
echo "Hello from the remote script!"
echo "Remote user is: $USER"
INNER_EOF

  chmod +x /tmp/remote_script.sh
  echo "Executing the new script on the remote host:"
  /tmp/remote_script.sh
OUTER_EOF
```

> **Good Practice:**

> Use redirections to manage input and output explicitly. Always redirect error messages to stderr (`>&2`) for proper separation from stdout.

Compound Commands

BASH offers numerous ways to combine simple commands to achieve our goals. We've already seen some of them in practice, but now let's look at a few more.

Bash has constructs called compound commands, which is a catch-all phrase covering several different concepts. We've already seen some of the compound commands Bash has to offer – `if` statements, `for` loops, `while` loops, the `[[` keyword, `case` , and `select` . We won't repeat that information again here. Instead, we'll explore the other compound commands we haven't seen yet: subshells, command grouping, and arithmetic evaluation.

In addition, we'll look at functions and aliases, which aren't compound commands but work in a similar way.

Subshells

A **subshell** is similar to a child process, except that more information is inherited. Subshells are created implicitly for each command in a pipeline. They are also created explicitly by using parentheses around a command:

```
 $ (cd /tmp || exit 1; date > timestamp)
$ pwd
/home/lhunath
```

When the subshell terminates, the `cd` command's effect is gone – we're back where we started. Likewise, any variables that are set during the subshell are not remembered. You can think of subshells as temporary shells.

Note that if the `cd` failed in that example, the `exit 1` would have terminated the subshell, but not our interactive shell. As you can guess, this is quite useful in real scripts.

Example:

```
 $ (var="temp"; echo $var)
temp
$ echo $var
# (nothing, var is unset in the parent shell)
```

Command Grouping

Commands may be grouped together using curly braces. Command groups allow a collection of commands to be considered as a whole with regards to redirection and control flow. All compound commands such as `if` statements and `while` loops do this as well, but command groups do only this. In that sense, command groups can be thought of as "null compound commands" in that they have no effect other than to group commands. They look a bit like subshells, with the difference being that command groups are executed in the same shell as everything else, rather than a new one. This is both faster and allows things like variable assignments to be visible outside of the command group.

All commands within a command group are within the scope of any redirections applied to a command group (or any compound command):

```
         $ { echo "Starting at $(date)"; rsync -av . /backup; echo "Finishing at $(date
>backup.log 2>&1
```

The above example truncates and opens the file `backup.log` on stdout, then points stderr at where stdout is currently pointing (`backup.log`), then runs each command with those redirections applied. The file descriptors remain open until all commands within the command group complete before they are automatically closed. This means `backup.log` is only opened a single time, not opened and closed for each command. The next example demonstrates this better:

```
         $ echo "cat
        mouse
        dog" > inputfile
        $ for var in {a..c}; do read -r "$var"; done < inputfile
        $ echo "$b"
        mouse
```

Notice how we didn't actually use a command group here. As previously explained, `for` being a compound command behaves just like a command group. It would have been extremely difficult to read the second line from a file without allowing for multiple `read` commands to read from a single open FD without rewinding to the start each time. Contrast with this:

```
         $ read -r a < inputfile
        $ read -r b < inputfile
        $ read -r c < inputfile
        $ echo "$b"
        cat
```

That's not what we wanted at all!

Command groups are also useful to shorten certain common tasks:

```
         $ [[ -f $CONFIGFILE ]] || { echo "Config file $CONFIGFILE not found" >&2; exit
```

The logical "or" now executes the command group if `$CONFIGFILE` doesn't exist rather than just the first simple command. A subshell would not have worked here, because the `exit 1` in a command group terminates the entire shell – which is what we want here.

Compare that with a differently formatted version:

```
         $ if [[ ! -f $CONFIGFILE ]]; then
        > echo "Config file $CONFIGFILE not found" >&2
        > exit 1
        > fi
```

If the command group is on a single line, as we've shown here, then there must be a semicolon before the closing `}` , (i.e., `{ ...; last command ; }`) otherwise, Bash would think `}` is an argument to the final command in the group. If the command group is spread across multiple lines, then the semicolon may be replaced by a newline:

```
         $ {
        >  echo "Starting at $(date)"
        >  rsync -av . /backup
```

```
>  echo "Finishing at $(date)"
> } > backup.log
```

Functions

Functions allow you to encapsulate a block of code for reuse. The syntax is:

```
 function_name() {
    # commands
}
```

Example:

```
 greet() {
    echo "Hello, $1!"
}
greet Alice
# Output: Hello, Alice!
```

Functions can take arguments (`$1` , `$2` , etc.) and return exit codes using `return` :

```
 check_file() {
    if [[ -f "$1" ]]; then
        echo "File $1 exists"
        return 0
    else
        echo "File $1 does not exist" >&2
        return 1
    fi
}
check_file config.txt
```

> **Good Practice:**
>
> Use functions to modularize code and improve readability. Name functions descriptively and use `return` for exit codes.

Function Control

- `return [N]` → exits a function or sourced script, setting exit status to N (default 0). Only valid inside functions or sourced scripts.
- `exit [N]` → terminates the entire script or shell with status N.

```
 myfunc() {
  [[ -z $1 ]] && return 1
  echo "Hello $1"
  return 0
}
myfunc "world"; echo $?
```

Destroying Constructs

Unsetting Variables and Functions

To remove a function or variable from your current shell environment use the unset command. It is usually a good idea to be explicit about whether a function or variable is to be unset using the -f or -v flags respectively. If unspecified, variables take precedence over functions.

$ unset -f myfunction

$ unset -v 'myArray[2]' # unset element 2 of myArray. The quoting is important to prevent globbing.

Terminating Processes with `kill`

While `unset` removes variables and functions from the shell's memory, the `kill` command is used to terminate running processes. Despite its name, `kill` is a general-purpose tool for sending *signals* to processes. A signal is a notification sent to a running program to inform it of an event.

The most common signals are:

Signal Name	Number	Description
`SIGTERM`	15	**Terminate (default)**. Gracefully requests a process to stop. The process can catch this signal and perform cleanup operations before exiting.
`SIGKILL`	9	**Kill**. Forcibly terminates a process immediately. The process cannot ignore this signal, so it's a last resort for unresponsive programs.
`SIGHUP`	1	**Hang Up**. Originally used to signal a terminal hang-up. It's now often used to tell a daemon to reload its configuration.
`SIGINT`	2	**Interrupt**. Sent when you press `Ctrl+C`. It interrupts the foreground process.
`SIGSTOP`	19	**Stop**. Pauses a process without terminating it. The process cannot ignore this signal.
`SIGCONT`	18	**Continue**. Resumes a process that was paused with `SIGSTOP`.

To use `kill`, you need the Process ID (PID) of the process you want to signal. You can find the PID using commands like `ps`, `pgrep`, or from the output of a backgrounded job.

Examples:

```
 # Start a process in the background
$ sleep 300 &
[1] 12345

# Send the default SIGTERM signal to PID 12345
$ kill 12345

# If it's still running, force kill it with SIGKILL
$ kill -9 12345
# or
$ kill -SIGKILL 12345

# You can also use job control syntax
```

```
$ sleep 300 &
[1] 54321
$ kill %1
```

You can use `pgrep` to find a PID by name:

```
 # Find the PID of the 'gedit' process and kill it
$ pgrep gedit
23456
$ kill 23456
# Or combine them
$ kill $(pgrep gedit)
```

Good Practice:

Always try to use `kill` (which sends `SIGTERM`) before resorting to `kill -9` (`SIGKILL`). This gives the program a chance to shut down cleanly, saving its state and releasing resources properly.

Sourcing

When you call one script from another, the new script inherits the environment of the original script, just like running any other program in UNIX. The environment includes the current working directory, open file descriptors, and environment variables, which you can view using the `export` command.

When the script that you ran (or any other program, for that matter) finishes executing, its environment is discarded. The environment of the first script will be the same as it was before the second script was called, although some of Bash's special parameters may have changed (such as the value of `$?`, the return value of the most recent command). This means, for example, you can't simply run a script to change your current working directory for you.

What you can do is to **source** the script, instead of running it as a child. You can do this using the `.` (dot) command:

```
 . ./myscript   # runs the commands from the file myscript in this environment
```

This is often called **dotting in** a script. The `.` tells Bash to read the commands in `./myscript` and run them in the current shell environment. Since the commands are run in the current shell, they can change the current shell's variables, working directory, open file descriptors, functions, etc.

Note that Bash has a second name for this command, `source`, but since this works identically to the `.` command, it's probably easier to just forget about it and use the `.` command, as that will work everywhere.

Example:

```
 $ cat myscript.sh
#!/usr/bin/env bash
var="Hello from myscript"
cd /tmp
$ . ./myscript.sh
$ echo $var
```

```
        Hello from myscript
        $ pwd
        /tmp
```

> **Good Practice:**
>
> Use sourcing when you need to modify the current shell's environment, such as setting variables or changing directories. Be cautious, as sourced scripts can overwrite existing variables or functions.

Job Control

Though not typically used in scripts, job control is very important in interactive shells. Job control allows you to interact with background jobs, suspend foreground jobs, and so on.

Theory

On POSIX systems, jobs are implemented as "process groups", with one process being the leader of the group. Each tty (terminal) has a single "foreground process group" that is allowed to interact with the terminal. All other process groups with the same controlling tty are considered background jobs and can be either running or suspended.

A job is suspended when its process group leader receives one of the signals `SIGSTOP` , `SIGTSTP` , `SIGTTIN` , or `SIGTTOU` . `SIGTTIN` (and `SIGTTOU` , if the user issued `stty tostop`) are automatically sent whenever a background job tries to read from or write to the terminal—this is why `cat &` is immediately suspended rather than running in the background.

Certain keypresses at the terminal cause signals to be sent to all processes in the foreground process group. These can be configured with `stty` , but are usually set to the defaults:

- **Ctrl-Z** sends `SIGTSTP` to the foreground job (usually suspending it).
- **Ctrl-C** sends `SIGINT` to the foreground job (usually terminating it).
- **Ctrl-*** sends `SIGQUIT` to the foreground job (usually causing it to dump core and abort).

Practice

Job control is on by default in interactive shells. It can be turned on for scripts with `set -m` or `set -o monitor` .

A foreground job can be suspended by pressing **Ctrl-Z**. There is no way to refer to the foreground job in Bash: if there is a foreground job other than Bash, then Bash is waiting for that job to terminate, and hence cannot execute any code (even traps are delayed until the foreground job terminates). The following commands, therefore, work on background (and suspended) jobs only.

Job control enables the following commands:

- `fg [jobspec]` : Bring a background job to the foreground.
- `bg [jobspec ...]` : Run a suspended job in the background.
- `suspend` : Suspend the shell (mostly useful when the parent process is a shell with job control).

Other commands for interacting with jobs include:

- `jobs [options] [jobspec ...]` : List suspended and background jobs. Options include `-p` (list process IDs only), `-s` (list only suspended jobs), and `-r` (list only running background jobs). If one or more jobspecs are specified, all other jobs are ignored.
- `kill` can take a jobspec instead of a process ID.
- `disown` tells Bash to forget about an existing job. This keeps Bash from automatically sending `SIGHUP` to the processes in that job, but also means it can no longer be referred to by jobspec.

Example:

```
 $ sleep 100 &
[1] 12345
$ jobs
[1]+  Running                 sleep 100 &
$ kill %1
[1]+  Terminated              sleep 100
```

So, what does all that mean? Job control allows you to have multiple things going on within a single terminal session. (This was terribly important in the old days when you only had one terminal on your desk, and no way to create virtual terminals to add more.) On modern systems and hardware, you have more choices available – you could, for example, run `screen` or `tmux` within a terminal to give you virtual terminals. Or within an X session, you could open more `xterm` or similar terminal emulators (and you can mix the two together as well).

But sometimes, a simple job control “suspend and background” comes in handy. Maybe you started a backup, and it’s taking longer than you expected. You can suspend it with **Ctrl-Z** and then put it in the background with `bg` , and get your shell prompt back, so that you can work on something else in the same terminal session while that backup is running.

Job Specifications

A job specification or “jobspec” is a way of referring to the processes that make up a job. A jobspec may be:

- `%n` : To refer to job number `n` .
- `%str` : To refer to a job that was started by a command beginning with `str` . It is an error if there is more than one such job.
- `%?str` : To refer to a job that was started by a command containing `str` . It is an error if there is more than one such job.
- `%%` or `%+` : To refer to the current job: the one most recently started in the background or suspended from the foreground. `fg` and `bg` will operate on this job if no jobspec is given.
- `%-` : For the previous job (the job that was `%%` before the current one).

It is possible to run an arbitrary command with a jobspec, using `jobs -x 'cmd args...'` . This replaces arguments that look like jobspecs with the PIDs of the corresponding process group leaders, then runs the command. For example, `jobs -x strace -p %%` will attach `strace` to the current job (most useful if it is running in the background rather than suspended).

Example:

```
 $ sleep 100 &
[1] 12345
$ jobs
```

```
[1]+  Running                 sleep 100 &
$ fg %1
sleep 100
^Z
[1]+  Stopped                 sleep 100
$ bg %1
[1]+  sleep 100 &
```

Practices

1. Choose Your Shell

The first thing you should do before starting a shell script, or any kind of script or program for that matter, is enumerate the requirements and the goal of that script. Then evaluate what the best tool is to accomplish those goals.

Keep in mind that BASH may not always be the best tool for the job.

If you need complex text processing, `awk`, `perl`, or `ruby` are generally well suited for the task, though the latter is rarely installed by default.

All of them can be given the code to interpret on the command line (`awk -- awk-script-code arguments` for the inline script, `perl/ruby -e perl/ruby-script-code -- arguments` for the inline script). As many of the characters in the syntax of those languages are also special in the syntax of the shell, you'll want to put the code argument inside single quotes. It's perfectly fine (and recommended for legibility) to write that code on several lines. You may also prefer to write separate `perl` / `ruby` scripts, but note that for `awk`, the script interface is broken by design as with a script starting with `#! /usr/bin/awk -f`, any argument passed to the `awk` script is considered for options for the `awk` interpreter first (not to mention the fact that `foo=bar` arguments are interpreted as variable assignments). That's fixed by the GNU implementation of `awk`, where you can use a `#! /usr/bin/gawk -E` shebang instead.

If you need to retrieve data from an HTML or XML file in a reliable manner, consider XPath/XSLT or a language that has a library available for parsing XML or HTML.

If you decide to use a shell script, evaluate the following questions:

- In the foreseeable future, might your script be needed in an environment where Bash is not available by default?
  - If so, then consider limiting the script to POSIX `sh` syntax only. POSIX defines a specification for a minimum standard language for `sh` and a few basic utilities, and virtually all systems supply a shell called `sh` (whether it is or is derived from Bash or the Bourne, Almquist, Forsyth, Korn, Yash, or Zsh shells) and the most common of those utilities (sometimes built-in to their `sh`) that try to conform to that standard. If you write your script to that standard, you can reasonably expect that script to run on any POSIX system. However, you will have to balance the need for portability against the cost of doing away with the more advanced features from shells such as Bash that you'd be unable to use.
- Are you considering supporting multiple operating systems to run the script? Then, not all the features found in the latest Bash will be available.
  - For example, in macOS, the default Bash version is permanently 3.2 due to licensing issues.
  - Some systems have 10 or 20 years supported lifespan and would keep the same version of

Bash during that span. For example, Solaris 10 was released in 2005 when the current version of Bash was 3.2 and will keep supplying that version (in an optional package) until its end of life in 2027.

If the above questions do not limit your choices, use all the Bash features you require, and then make note of which version of Bash is required to run your script.

Using Bash means you can use arrays and avoid some awkward `sh` scripting techniques.

> **Good Practice:**
>
> - Examine scripting examples on the Web carefully before giving in to the temptation to use their code or other constructs. Some may be broken or may not work universally.
> - Remember to use the correct shebang. For Bash scripts, write `#!/usr/bin/env bash` at the top of your script. Omitting this or using the `#!/bin/sh` header is incorrect. The `sh` limits the script to run under the POSIX shell features, even if the operating system's `/bin/sh` is a symlink to Bash, as is the case in Enterprise Linux or a special build of Bash like on macOS.
> - In Bash scripts, consider using the `[[...]]` construct over the `[` command unless you're doing decimal integer comparisons. Bash's `[[...]]` is more forgiving in many ways, like safeguarding against word splitting or globbing when you forget to quote variables. Its specific micro-syntax also supports glob pattern and regular expression matching capabilities.
> - Skip over the over-30-years-old obsolete command substitution \ …`` syntax. The modern POSIX version $(…)` is more consistent with the syntax of Expansion, supports multiple quoting, and is easier to nest. An example: `fullpath=$(CDPATH= cd -P – "$(readlink -- "$dir")" && pwd)`
> - First and foremost, remember to "use more quotes". This will protect the strings and parameter expansions from word splitting and globbing.

2. Debugging and Robustness

When writing Bash scripts, debugging and ensuring robustness are critical to producing reliable and maintainable code. Here are some practices to follow:

> **Good Practice:**
>
> - **Use `set` Options for Debugging**: Bash provides several options to help debug scripts. Enable these by adding them at the top of your script or using them interactively:
>   - `set -x` : Prints commands and their arguments as they are executed, useful for tracing what your script is doing.
>   - `set -e` : Exits the script immediately if any command returns a non-zero exit status, preventing further execution after an error.
>   - `set -u` : Treats unset variables as an error and exits immediately, helping catch typos or undefined variables.
>   - `set -o pipefail` : Ensures that a pipeline's exit status is non-zero if any command in the pipeline fails, not just the last one.
>
>   Example:
>
>   ```
>   #!/usr/bin/env bash
>   ```

```
set -euxo pipefail
echo "This is a test"
ls /nonexistent  # This will cause the script to exit due to set -e
echo "This won't run"
```

These options do not have to be set for the entire script. You can toggle them on and off to debug only specific sections. Use `set -x` to enable tracing and `set +x` to disable it. This can be invaluable for pinpointing issues without flooding your output.

Example of toggling:

```
echo "This part is not traced."

# Turn on debugging for a specific block of code
set -x

a=1
b=2
if (( a < b )); then
  echo "$a is less than $b"
fi

# Turn off debugging
set +x

echo "And this part is not traced either."
```

- **Use Traps for Cleanup**: The `trap` command allows you to execute code when certain signals are received or when the script exits. This is useful for cleaning up temporary files or restoring states.

  Example:

  ```
  trap 'rm -f /tmp/tempfile; echo "Cleaned up"; exit' EXIT
  touch /tmp/tempfile
  echo "Doing something"
  ```

- **Quote Everything**: Always quote parameter expansions, command substitutions, and strings to prevent word splitting and globbing issues. This ensures your script handles filenames and data with spaces or special characters correctly.

  Example:

  ```
  file="my file.txt"
  ls "$file"   # Correct: preserves spaces
  ls $file     # Wrong: splits into "my" and "file.txt"
  ```

- **Test Your Scripts**: Before deploying a script, test it thoroughly with edge cases, such as empty inputs, files with spaces, or special characters. Use tools like `bash -n script.sh` to check syntax without executing the script.

- **Avoid Common Pitfalls**:

  - Don't parse `ls` output, as it's unreliable for filenames with spaces or newlines. Use globs or `find` instead.

- Avoid unquoted variables in conditions like `[ $var = something ]`, as it fails if `$var` is unset. Use `[[ $var == something ]]` instead.
- Be cautious with redirections; ensure files exist before writing to them to avoid overwriting important data.

Good Practice:

Always include `set -euo pipefail` at the start of your scripts to catch errors early. Quote all expansions unless you explicitly need word splitting. Test scripts with edge cases to ensure robustness.

3. Script Structure

Organizing your script logically improves readability and maintainability. Follow these guidelines:

Good Practice:

- **Use Functions**: Break your script into reusable functions to modularize code. Functions make scripts easier to maintain and test.

  Example:

  ```
  log() {
      echo "[$(date)] $*" >&2
  }
  log "Starting script"
  ```

- **Comment Generously**: Add comments to explain the purpose of complex logic, functions, or non-obvious commands. Use clear, concise comments to avoid clutter.

  Example:

  ```
  # Backup all .jpg files to /backup
  cp -v *.jpg /backup
  ```

- **Consistent Formatting**: Use consistent indentation (e.g., 2 or 4 spaces) and follow a style guide for variable names, function names, and command layout. For example, use lowercase for variable names and descriptive names for functions.
- **Exit Codes**: Return meaningful exit codes from your script. Use `exit 0` for success and non-zero values (e.g., `exit 1`) for specific errors. Document what each exit code means.

  Example:

  ```
  if [[ ! -f config.txt ]]; then
      echo "Error: config.txt not found" >&2
      exit 1
  fi
  ```

- **Shebang Line**: Always include `#!/usr/bin/env bash` at the top of your script to ensure it runs with Bash, not a POSIX shell, unless portability to `sh` is required.

Good Practice:

Structure your script with a clear flow: initialization (variables, options), functions, and main logic. Use comments to separate sections and explain intent. Always return explicit exit codes.

4. Error Handling

Proper error handling ensures your script fails gracefully and provides meaningful feedback:

Good Practice:

- **Check Command Success**: Use control operators (`&&` , `||`) or check `$?` to handle command failures.

  Example:

  ```
  cp file.txt /backup || { echo "Copy failed" >&2; exit 1; }
  ```

- **Validate Input**: Check command-line arguments, environment variables, and file existence before proceeding.

  Example:

  ```
  if [[ $# -eq 0 ]]; then
      echo "Error: No arguments provided" >&2
      exit 1
  fi
  ```

- **Redirect Errors**: Send error messages to stderr using `>&2` to separate them from normal output.

  Example:

  ```
  echo "Error: Something went wrong" >&2
  ```

- **Use `trap` for Unexpected Errors**: Catch signals like `SIGINT` (Ctrl+C) to handle interruptions gracefully.

  Example:

  ```
  trap 'echo "Script interrupted"; exit 1' INT
  ```

Good Practice:

Always validate inputs and check for errors after critical commands. Use `trap` to handle cleanup and interruptions. Direct error messages to stderr.

5. Portability Considerations

If your script needs to run on multiple systems, consider portability:

Good Practice:

- **Stick to POSIX Features**: If Bash-specific features (e.g., arrays, `[[`) aren't needed, write POSIX-compliant scripts using `#!/bin/sh`. This ensures compatibility with systems lacking Bash or using older versions.
- **Check Tool Availability**: Some commands (e.g., `readlink`, `stat`) may not exist or behave differently on non-Linux systems. Test on target platforms or use alternatives like `find`.
- **Avoid Bashisms**: Features like `[[`, arrays, or process substitution are Bash-specific. Use tools like `checkbashisms` to identify non-POSIX constructs.

  Example:

  ```
   # POSIX-compliant test instead of [[
  if [ ! -f file.txt ]; then
      echo "File not found"
  fi
  ```

Good Practice:

Decide early if portability is needed. If targeting only Bash environments, use its features freely but note the required Bash version. For maximum portability, stick to POSIX `sh` and test across platforms.

6. Security Considerations

Writing secure Bash scripts is crucial, especially for scripts running with elevated privileges:

Good Practice:

- **Avoid `eval`**: The `eval` command executes arbitrary strings as code, which can be dangerous if user input is involved. Use safer alternatives like arrays or functions.

  Example (avoid):

  ```
  eval "command $user_input"  # Dangerous!
  ```

- **Sanitize Input**: Validate and sanitize user input to prevent injection attacks. Use `case` or pattern matching to restrict acceptable values.

  Example:

  ```
  case $input in
     [a-zA-Z]*) echo "Valid input" ;;      *)
       echo "Invalid input" >&2
       exit 1
       ;;  esac
  ```

- **Use Absolute Paths**: Avoid relying on `$PATH` for commands in scripts, as it can be manipulated. Use absolute paths for critical commands.

Example:

```
 /usr/bin/cp file.txt /backup  # Safe
cp file.txt /backup           # Risky if PATH is modified
```

- **Limit Permissions**: Run scripts with the least privileges necessary. Avoid running as root unless absolutely required.
- **Quote Variables**: Prevent command injection by quoting variables, especially those containing user input.

Good Practice:

Never use `eval` with untrusted input. Always quote variables and use absolute paths for commands. Validate all inputs and run with minimal permissions.

7. Performance Considerations

For scripts processing large datasets or running frequently, optimize for performance:

Good Practice:

- **Minimize External Commands**: Each external command (e.g., `grep` , `sed`) spawns a new process, which is slow. Use Bash builtins where possible.

  Example:

  ```
   # Slow: uses external command
  if echo "$var" | grep -q pattern; then
      echo "Found"
  fi
  # Faster: uses Bash pattern matching
  if [[ $var == *pattern* ]]; then
      echo "Found"
  fi
  ```

- **Use Arrays for Loops**: When processing multiple items, store them in arrays rather than parsing strings.

  Example:

  ```
   files=( *.txt )
  for file in "${files[@]}"; do
      echo "Processing $file"
  done
  ```

- **Avoid Unnecessary Subshells**: Subshells (e.g., `(command)`) create new processes, which are costly. Use command groups `{ command; }` when possible.
- **Batch Operations**: Combine operations to reduce I/O or command invocations.

Example:

```
# Slow: multiple writes
for i in {1..10}; do echo "$i" >> file; done
# Faster: single write
printf '%s\n' {1..10} > file
```

Good Practice:

Prefer Bash builtins and arrays over external commands and string parsing. Minimize subshells and batch operations where possible.

8. Documentation and Maintenance

Well-documented scripts are easier to maintain and share:

Good Practice:

- **Include a Header**: Add a comment block at the start of your script with its purpose, usage, and dependencies.

  Example:

  ```
  #!/usr/bin/env bash
  # Description: Backs up all .jpg files to /backup
  # Usage: ./backup.sh [source_dir]
  # Dependencies: rsync
  ```

- **Version Control**: Use a version control system (e.g., Git) to track changes and collaborate.
- **Log Actions**: Log significant actions to a file or stderr for debugging and auditing.

  Example:

  ```
  echo "$(date): Starting backup" >> /var/log/backup.log
  ```

- **Update Regularly**: Revisit scripts to update for new Bash versions or system changes. Check for deprecated features or better alternatives.

Good Practice:

Include a clear header with usage instructions. Use version control and logging to track changes and actions. Keep scripts updated to maintain compatibility.

Variable Scoping

- By default, variables in Bash are GLOBAL. Any variable declared outside functions is accessible everywhere unless shadowed.
- Functions can read/write global vars. If you want isolation, use `local` inside functions.

- `local var=value` → variable exists only inside the function (and its children).
- Subshells: `( ... )` run in a new process. Variable changes inside do not affect the parent shell.
- Exported vars: `export VAR=value` makes variable available to child processes, but not parents.
- Sourced scripts (`source file.sh` or `. file.sh`) run in the current shell, so variables persist. Executed scripts (`bash file.sh`) run in a new process, so vars do not persist.

```bash
 x=10
foo() {
  local x=99       # local variable shadows global
  echo "Inside foo: $x"
}
foo
echo "Outside foo: $x"   # still 10
```