Bash Mastery

The complete guide to BASH shell scripting

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
August 3, 2023
United States

Contents

1 Setting up scripts

In order to write shell scripts, we must use the file extension .sh. When we begin the script, we must include a *shebang*, which looks something like:

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

However, this could depend on the users system. To locate which file path to use, we can use the command:

```
which bash
```

This will provide the path to use for the shebang.

The anatomy of a shell script can be described with the following parts:

1. Shebang
2. Commands
3. Exit statement (0=successful, 1-255=unsuccessful)

Example:

```
#!/usr/bin/bash

echo "Hello World!"
exit 0
```

Note:-

The recommended file permissions for scripts is 744 (chmod 744 *filename*)

1.1 Adding scripts to PATH

To add scripts to your PATH for BASH, we can open up our .bashrc, and add at the bottom of the file:

```
export PATH="$PATH:$HOME/dirlocation"
```

This will append some directory to the end of our PATH variable. To do the same for the *fish* shell, in the fish config file, we can add.

```
set -gx PATH $PATH $HOME/somedirectory
```

2 Variables and Shell Expansions

2.1 User-Defined variables and parameter expansion

Definition 1:

A **parameter** is any entity that stores values. In bash, we have three types:

1. Variables
2. Positional Parameters
3. Special Parameters

To define variables in our script, we can do:

```
identifier=value # NO WHITESPACE
name="nate" # Example
declare -i a=1 # Integer variable

# Parameter Expansion (Reference variables)
echo "Hello, ${name}!"
```

2.2 Shell variables

Shell variables are builtin variables that we can access but don't need to define ourself, some common shell variables are:

- PATH
- HOME
- USER
- HOSTNAME
- HOSTTYPE

2.3 Positional Parameters

Positional parameters are variables that hold the command-line arguments to a script or function. They are denoted by numbers.

1. $0 (Contains the name of the script)
2. $1, $2 ... $n (The first, second, third, etc. arguments to the script or function.)
3. $# (The number of arguments passed to the script or function.)
4. $@ (All the arguments. When quoted ("$@"), it treats each argument as a separate word. Useful for loops, more on this later)
5. $* (All the arguments. When quoted ("$*"), it treats all arguments as a single word. Useful for loops, more on this later)

2.4 Special Parameters

These are variables that provide special functionality or information about the script or command's execution:

- $?: The exit status of the last executed command. 0 usually indicates success, and a non-zero value indicates an error.
- $$: The process ID (PID) of the currently executing script or shell instance.
- $!: The process ID (PID) of the last backgrounded command.
- $-: The current options set for the shell. For instance, if you used `set -x` for debugging, `x` would be part of the value.
- $_: The last argument of the previous command. Also sometimes used to get the last path argument to the `cd` command.

2.5 Parameter Expansion Tricks

Default Values:

- ${parameter:-word}: If parameter is unset or null, this expansion will return word. Otherwise, it returns the value of parameter.
- ${parameter:=word}: If parameter is unset or null, it will be set to word.

String Length:

- $#parameter: Returns the length of the value of the parameter.

Substring Expansion:

- ${parameter:offset:length}: Extracts a substring from $parameter starting at offset (0-indexed) and of length length.

String Removal (Pattern Matching):

- ${parameter#pattern}: Removes the shortest match of pattern from the beginning of $parameter.
- ${parameter##pattern}: Removes the longest match of pattern from the beginning of $parameter.
- ${parameter%pattern}: Removes the shortest match of pattern from the end of $parameter.
- ${parameter%%pattern}: Removes the longest match of pattern from the end of $parameter.

String Replacement:

- ${parameter/pattern/string}: Replaces the first match of pattern with string in $parameter.
- ${parameter//pattern/string}: Replaces all matches of pattern with string in $parameter

Variable Indirection:

- ${!parameter}: Treats the value of parameter as the name of another variable, and fetches the value of that variable.

Case Modification:

- ${parameter^}: Capitalizes the first letter of the value.
- ${parameter^^}: Capitalizes all letters of the value.
- ${parameter,}: Converts the first letter to lowercase.
- ${parameter,,}: Converts all letters to lowercase.

2.6 Command Substitution

Concept 1: Command substitution can be used to:
□

- Save the output of commands in variables
- Use the output of one command *inside* another command

The syntax for this is:

```
$(command)
# Example...
time=$(date +%H:%M:%S)
echo "Hello, the current time is ${time}"
```

2.7 Arithmetic Expansion

The syntax for *Arithmetic Expansion* is:

```
$((expression))
# Example...
echo $((1+1)) # 2


x=1
y=1
echo $(( x + y ))
```

2.8 Dealing with floating point numbers

To be able to do floating point arithmetic in our scripts, we need to use the **bc** command.

Example:

```
echo "scale=2; 5/2" | bc # 2.50
result=$(echo "scale=2; 5/2" | bc)
# scale sets the precision of the output
```

2.9 Tilde Expansion

I'm sure you're already familiar with using tilde to jump to your home directory, but we can also use ∼- to jump between our current directory, and our home directory

2.10 Brace Expansion

We have two types of brace expansions:

- String lists
- Range lists

Here is examples of what we can do with brace expansion:

```
1 echo {jan,feb,march} # jan feb march NO WHITESPACE IN BRACES
2 echo {1..5} # 1 2 3 4 5
3 echo {1..10..2} # 1 3 5 7 9
4 echo {a..e} # a b c d e
5 echo {a,b}{1,2,3} # (Cartesian product...) a1 a2 a3 b1 b2 b3
6 # Useful for commands...
7 mkdir dir_{1..3}.txt
8 touch file_{1..5}.txt
```

3 How Bash Processes Commands

Bash uses a 5 step process to interpret a command

1. **Tokenisation:** A token is a sequence of characters that is considered as a single unit by the shell. The shell determines were a token starts and ends with the following special (meta) characters

   - |
   - &
   - ;
   - ()
   - < >
   - Space, tab, newline

   The shell then determines whether these tokens are words, or operators. A **word** is a token that does not contain an unquoted metacharacter. **Operators** are tokens that contain at least one unquoted metacharacter. This makes quoting a key concept in how the shell operates

2. **Command identification:** The shell breaks the command up into either simple, or compound commands. **Simple commands** are just a bunch of individual words, and each simple command is terminated by a control operator. **Compound commands** provide bash with its programming constructs, such as if statements, for loops, while loops, etc...

3. **Expansions:**

4. **Quote removal:** We add quotes to control how the command is interpreted, so this step will simply remove all those supportive quotes.

5. **Redirection:**

After these 5 steps are completed, bash will then execute the command that is left over.

3.1 Quoting

Concept 2: Quoting is about **removing special meanings**. There are three types of quoting:
□

- **Backslash (\):** This removes special meaning from next character
- **Single Quotes:** Removes special meaning from all characters inside
- **Double Quotes:** Removes special meaning from all inside except dollar signs ($) and backticks (`)

3.2 Step 1. Tokenisation

We learned earlier that whether or not a token is interpreted as a word or an operator, depends on if there are any **unquoted metacharacters**. In bash we have two types of operators:

Control operators:

- Newline: command separator, similar to the semicolon (;).
- | used to send the output of one command as the input to another command.
- || Used to execute the command following it only if the command preceding it fails
- & This is used to execute a command in the background
- && Used to execute the command following it only if the command preceding it succeeds
- ; Acts as a command separator.
- ;; Used in the context of a case statement in shell scripting. It signifies the end of an option within a case block.
- ;& Also used in a case statement. After executing the associated block for a matched pattern, the control will flow to the block of the next pattern without testing.
- ;;& Another operator used in a case statement. The control will test the next pattern after executing the block for the matched pattern.
- |& This is shorthand for 2>&1 |. It redirects both standard output (stdout) and standard error (stderr) of the command before the pipe to the command after the pipe.
- (used to group commands and execute them in a subshell.
-) used to group commands and execute them in a subshell.

Redirection Operators

- < Redirects input for a command from a file rather than from the keyboard.
- > Redirects the output of a command to a file
- « Provides multiple lines of input to a command
- » Similar to >, but instead of overwriting the file, it appends to the file.
- <& Duplicates one input file descriptor to another, allowing for more advanced redirections.
- >& Duplicates one output file descriptor to another.
- >| (Clobber): This is used in conjunction with the noclobber option in Bash (set -o noclobber). It allows you to forcefully overwrite a file when output redirection is used, even if noclobber is set.
- «- Similar to «, but it allows leading tabs (not spaces) to be ignored,
- <> Opens a file in read-write mode for a command.

3.3 Step 2. Command Identification (Simple commands)

As we stated earlier, we have two types of commands, simple, and complex. Let's have a look at a simple command.

$$\underbrace{\text{echo}}_{\substack{\text{Command}\\ \text{name}}} \quad \underbrace{1\ 2\ 3}_{\substack{\text{individual}\\ \text{arguments}}}$$

All simple commands are terminated by a control operator, which we have discussed earlier. In this case, it is a newline

Thus, "echo" is identified as the command, and the rest is identified as the commands arguments, since there are no **control operators**

3.4 Step 2. Command Identification (Simple commands)

Concept 3: Compound commands are essentially bash's programming constructs. Each compound command begins with a **reserved word** and is terminated by a **reserved word**. □

3.5 Step 3. Expansions

The shell goes through 4 stages of expansion.

- **Stage 1:** Brace expansion
- **Stage 2:**
  - Parameter expansion
  - Arithmetic expansion
  - Command substation
  - Tilde expansion
- **Stage 3:** Word splitting
- **Stage 4:** Globbing

An important thing to know is that expansions in earlier stages, happen before expansions in later stages. This means if we had the code:

```
1 x=10
2 echo {1..$x}
```

We would **not** get the expected results, this is because the brace expansion will take place **before** the parameter expansion.

Another important thing to know is that items in **stage 2** have the same precedence, thus, they will be preformed left to right. Similar to how multiplication and division works in the rules of PEMDAS.

3.6 Word splitting

Concept 4: Word splitting is a process the shell performs to split the result of some unquoted expansions into separate words. Word splitting can have some very significant effects on how your commands are interpreted
□

Word splitting is only performed on the results of unquoted

- Parameter expansions
- Command substitutions
- Arithmetic expansions

The characters used to split words are governed by the IFS (Internal Field Separator) variable.

- Space, tab, and newline

Suppose we have:

```
1 numbers="1 2 3 4 5"
2 touch ${numbers}
```

We will get 5 **different** files, labeled 1-5. This is because the parameter expansion was **unquoted**. Instead, we can do:

```
1 numbers="1 2 3 4 5"
2 touch "${numbers}"
```

This will prevent word splitting, and treat ”1 2 3 4 5” as a single word. Creating just **one** file.

So we shall follow one simple rule. If we want the output of a:

- Parameter expansion
- Command substation
- Arithmetic expansion

To be considered as a **single word**, we must **wrap that expansion in double quotes!**

3.7 Globbing

Concept 5: Globbing is used as shorthand for listing the files that a command should operate on. Globbing is **only** performed on words (not operators)
□

Globbing patters are words that contain unquoted **Special pattern characters**

- * (used as a wildcard for any word)
- ? (used as a wildcard for any character, very strict in terms of length)
- ⟦ (Like question mark, but we can control the characters)

```
ls *.txt # will list all .txt files
ls file?.txt # will list all .txt files named "file[somecharacter]"
# Note: Anything with 2 or more characters after "file" will not be
↪  listed
ls file[ab].txt # will list either filea.txt, fileb.txt, or nothing
```

We can also treat the brackets as character classes...

```
ls file[a-z].txt
```

3.8 Quote removal

During the quote removal stage, the shell removes ALL unquoted backslashes, single quote characters, and double quote characters that **did not** result from a shell expansion.

3.9 Redirection

Data Streams

- Stream 0: Standard input (stdin)
- Stream 0: Standard output (stdout)
- Stream 0: Standard error (stderr)

Standard input provides us with an alternative way of providing input to a command, aside from using command line arguments. Default source is keyboard

Standard output contains the data that is produced after a successful command execution. Default source is terminal

Standard error contains all error messages and status messages that a command produces . Default source is terminal

Using redirection is all about changing the sources or destinations of these streams. For example, if we use the *cat* command, we are changing the input stream source from the keyboard to a file.

For redirection we can use the following operators:

- > Redirects standard output to a file, overwriting the file if it exists.
- » Redirects standard output to a file, appending to the file if it exists.
- < Takes input from a file
- 2> Redirects standard error to a file, overwriting the file if it exists.
- 2» Redirects standard error to a file, appending to the file if it exists.
- &> or >& Redirects both standard output and standard error to a file, overwriting the file if it exists.
- &» Redirects both standard output and standard error to a file, appending to the file if it exists.
- | (pipe) Takes the standard output of one command and uses it as the standard input for another command.

4 Requesting user input

4.1 Positional Parameters

Recall from 2.3, where we discussed the concept of **positional parameters**, in this section, we will discuss how we can use them in our scripts.

Suppose we had some script call

```
./1.sh nate $HOME
```

Here we are passing two arguments to this script. For these arguments to be relevant, let's take a look at the script...

```
echo "Hello, I am ${0}"
echo "My name is, ${1}"
echo "My home directory is ${2}"
# Hello, I am ./1
# My name is, nate
# My home directory is /home/datura
```

So you can see we are making use of some **positional parameters** to gain access to the things that we passed to the script we it was executed.

Note:-

It's important we use the curly brace expansion syntax for double digit numbers or we might get unexpected results

4.2 Special Parameters

Now that we have learned how to use **positional parameters** in our scripts. We will take a look at **Special parameters**, these parameters give us information about the script. The definitions for these were described in 2.4

```
echo "Exit code: ${?}"
echo "PID: ${??}"
# Exit code: 0
# PID: 0
```

4.3 The read command

We can use the **read** command in bash to get input from the user, for example:

```
read name age
# Or
read -p "Enter your name: " name
read -p "Enter your age " age
```

Then when we run this script, we can enter some data. This data will be stored in the variables ”name” and ”age”. Notice that the second example lets us define a prompt.

Other options we have for the read command are:

- -t: timeout, specified in seconds
- -s: make input s.t it is not visible in the terminal

4.4 The select command

Using the **select** command, we can create menus for the user, and have the shell carry out different commands based on what is selected.

```
PS3="Enter: "
select day in a b c d e f; do
    echo "You entered ${day}"
break
done
```

There are some key things to notice here. For one, we can define the **PS3** variable to give the user a custom prompt. Second, this select command will by default result in a continuous loop. Thus, we must use the **break** command to terminate it.

5 Logic

5.1 Chaining commands

In bash we have the following operators:

- & (send command to background)
- ; (chain commands)
- && (only execute next command if previous was successful)
- || (only execute next command if previous was unsuccessful)

5.2 Test commands and conditional operators

Concept 6: If a test is evaluated to be true, the test will return an exit status of 0. If a test is evaluated to be false, the test will return an exit status of 1. The **test** command can be used in bash to compare different pieces of information.
□

The syntax for the test command is a set of square brackets [ ]. Its important that you have a space separating the contents of the square bracket and the actual square brackets otherwise you will get a syntax error.

Note:-

The set of square brackets is the **short hand form**, the long hand form is using the **test** keyword

Example:

In conjunction with the test command, we can use the following operators:

File Tests:

- -s FILE: True if FILE exists and has a size greater than zero.
- -h FILE or -L FILE: True if FILE exists and is a symbolic link.
- -r FILE: True if FILE exists and is readable.
- -e FILE: True if FILE exists.
- -d FILE: True if FILE exists and is a directory.
- -w FILE: True if FILE exists and is writable.
- -x FILE: True if FILE exists and is executable.
- -f FILE: True if FILE exists and is a regular file.

String tests:

- -z STRING: True if STRING has a length of zero.
- -n STRING: True if STRING has a length greater than zero.
- STRING1 = STRING2: True if STRING1 and STRING2 are the same.
- STRING1 != STRING2: True if STRING1 and STRING2 are not the same.

Arithmetic Comparisons:

- -eq: Equal to.
- -ne: Not equal to.
- -lt: Less than.
- -le: Less than or equal to.
- -gt: Greater than.
- -ge: Greater than or equal to

Logical Operators:

- ! (logical not)
- -a (logical and)
- -o (logical or)

Example:

```
# Long hand
if test 1 -le 2; then
    echo "1 is less than 2"
fi
# Short hand...
if [ 1 -le 2 ]; then
    echo "1 is less than 2"
fi
```

Additionally, we have the **double bracket**. This is known as the extended test command, offers several advantages and additional capabilities over the traditional test command. It allows for:

- Word splitting does not occur
- Regex matching using =∼
- String comparison using == or !=
- Compound comparisons using && (and) and || (or)

5.3 If statements

The syntax for if statements

```
if [ condition ]; then

elif [ condition ]; then

else

fi
```

5.3.1 Double Brackets (Extended test)

The double bracket is a more modern syntax introduced by the KornShell (ksh) and later adopted by bash. It provides enhanced functionality over single brackets.

1. **Enhanced Features:** Supports additional features like pattern matching with ==, regex matching with = , and logical operators and ||.
2. **No Word Splitting:** Inside [[...]], word splitting and pathname expansion (globbing) are not performed.
3. **Safer:** Reduces the need for excessive quoting, minimizing common errors with spaces and special characters.
4. **Non-POSIX:** Not as portable as single brackets, primarily available in bash, ksh, and zsh.

```
if [[ "$1" -eq 10 ]]; then
echo "The number is 10"
fi

if [[ "$string" == "pattern*" ]]; then
echo "String matches pattern"
fi

if [[ "$string" =~ ^regex$ ]]; then
echo "String matches regex"
fi
```

5.3.2 Pattern matching with ==

- The == operator is used for string comparison within double brackets.
- When the right-hand side of == contains wildcard characters like *, it performs pattern matching.

Note:-

when using == with double brackets, you are limited to basic pattern matching with wildcards such as *.

5.3.3 Regex matching with =~

- The = operator is used for regular expression (regex) matching within double brackets.
- The right-hand side of = is treated as a regex pattern.

5.3.4 Arithmetic evalutations in if statements

The ((...)) construct is used to evaluate arithmetic expressions. It allows you to perform arithmetic operations and evaluate the result directly within the script. When used inside an if statement, it evaluates the expression and returns a boolean value (true or false) based on the result of the expression.

```
if (( 9 % 3 == 0 )); then
    echo "9 is divisible by 3"
fi
```

5.4 Negation of a test

```
if ! [[ ... ]]; then
    ...
fi
```

Note:-

Its important to note if we had multiple tests in the same if block we would need to negate each one

```
if ! [ ... ] || ! [ ... ]; then
    ...
fi
```

5.5 Case statements

Syntax:

```
case "${var}" in
    value)
        statement
        ;;
    value)
        statement
        ;;
esac
```

6 Process options and read from files

6.1 While loops

Syntax:

```
while [[ test ]]; do
    statements;
done
```

So for example:

```
read -p "Enter a number: " num
while [[ $num -lt 5 ]]; do
    echo $num
    num=$(($num+1))
done
```

6.2 Handling command line options

For this we can use the **getopts** command to get any options that the user runs with the script

Syntax:

```
getopts "ab" varname # Defines two possible options, 'a', and 'b',
↪  stores in varname
getopts 'a:b:' varname # Lets the options take in values. Stores in
↪  varname
echo $OPTARG # Built in variable to house the option value
```

In order to ensure that the getopts command gets ALL options, we must write the command as the test for a while loop

```
while getopts 'a:b:' varname; do
    echo $OPTARG
done
# Run script
./scriptname -a 2 -b 3
```

6.3 Iterating over files with read-while loop

Concept 7: Read-while loops are while loops that use the **read** command as their test command
□

```
while read line; do
    echo "$line"
done < "${1}"
```

We can also use a technique called **process substation** to redirect standard input from a process

```
while read line; do
    echo "$line"
done < <(ls $HOME)
```

7 Arrays and For loops

7.1 Working with indexed arrays

Arrays in bash are as follows...

```
myarr=(1 2 3)  # Direct assignment
declare -a myarr=(1 2 3) # Typeset
myarr[0] # Standard indexing
myarr[0]=1 # Standard reassigning
read -a myarr <<< "1 2 3" # nonnatural assignment

# Changing IFS for read command
IFS="${IFS} ," # add colon to IFS
read myarr2 >>> "1,2,3"

# Outputting values
echo ${myarr[0]}
echo ${myarr[@]} # Output all values
echo ${#myarr[@]} # Length of array
echo ${!myarr[@]} # Output index numbers

# Appending to array
myarr+=(1) # Add to end
```

To summarize, we have:

- Declaring arrays (direct, typeset, read)
- Indexing
- Reassign
- Read with Custom IFS
- Outputting (specific, all, length, index numbers)
- Appending

7.2 readarray command

We can use the readarray command to create arrays from lines in a file

```
readarray arr < file.txt
echo "${arr[@]@Q}" # Show raw data (trailing newlines)
readarray -t arr < file.txt # Not storing trailing newlines

# Alternatively...
IFS=
while read -r line; do
    arr+=($line)
done < file.txt
```

8 For loops in bash

We have a couple different ways of creating for loops in bash.

8.1 Standard Syntax

```
for var in list-of-values; do
    statements
done

# Example
for var in 1 2 3 4; do
    echo $var
done
```

8.2 C-style syntax

```
for (( initilization; condition; update )); do
    statements
done
```

8.3 Using the seq command

```
for i in $(seq start end); do
    statements
done
```

8.4 Iterating over arrays

```
arr=(1 2 3)
for i in "${arr[@]}"; do
    echo $i
done
```

9 Scheduling and automation

9.1 The ”at” command

Concept 8: The ”at” command is used to schedule a command or script to run once at a specified time in the future. It's the most basic way to schedule commands in linux
□

By default, we will not have access to this command. Thus, we need to install the package

```
sudo pacman -Sy at
sudo apt install at
```

This we give us access to the **atd** (at daemon).

```
sudo systemctl status atd
```

The usage for this command is:

```
at <time>
```

We can list the jobs that are scheduled with *at -l*. We can remove jobs with *at -r <job-id>*. To schedule scripts, can add the -f option. For example *at <time> -f myscript.sh*.

We can also give specific days or dates. For example:

```
at 9:00am Monday -f myscript.sh
at 9:00am 12/23/2023 -f myscript.sh
at 9:00am next week -f myscript.sh
at now + 5 [seconds/minutes/days/etc]
```

9.2 cron

After we install and start the cron service, we can then access its features. To edit the crontab file, we can run:

```
crontab -e
```

The usage for this service is pretty straight forward, we can just follow the documentation found in the crontab file to schedule our scripts.

Note: The format is minute hour "day of month" "month number" "day of week". We can use the wildcard (*) in place of specific times or days to signify perpetuity.

9.3 Crontab.guru

Crontab.guru is a website that can generate cron expressions for us

9.4 cron directories

Concept 9: Cron directories are directories on your system where you can place scripts to run at a particular frequency. If we run:
□

```
ls /etc | grep cron
```

We can then enter /etc/crontab to see how these directories are run and timed. These schedules use the **run-parts** command to run all executable files within any of the directories

9.5 anacron

The difference between this program and the regular **cron** program is that anacron can recover missed jobs. Thus, if the machine is turned off when a job is supposed to be activated, this program will run any missed jobs when the machine is turned on

The anacron file is located /etc/anacron, and anything scheluding in the cron directories will be taken care of by anacron. The syntax for the anacron file is

```
frequency interval_between_scripts (minutes) reference command
```