Bash Mastery

The complete guide to BASH shell scripting

Nathan Warner

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
# When dealing with arithmetic expansion, we do not need a $ to reference variables
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
# 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:

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

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

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:

```
x=10
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:

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

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

```
numbers="1 2 3 4 5"
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