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. 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 Command Lines