dave eddy <bahamas10> (/)

All Dropped Packets go to Heaven

Dark Mode

Bash Style Guide

This style guide is meant to outline how to write bash scripts with a style that makes them safe and predictable. This guide is based on this wiki (http://mywiki.wooledge.org), specifically this page:

http://mywiki.wooledge.org/BashGuide/Practices

If anything is not mentioned explicitly in this guide, it defaults to matching whatever is outlined in the wiki.

Fork this style guide on GitHub https://github.com/bahamas10/bash-style-guide

Preface

I wrote this guide originally for a project I had worked on called Basher (https://github.com/bahamas10/basher). The idea was to make a program like Puppet (https://puppet.com/) or Chef (https://www.chef.io/) but using nothing but Bash - simple scripts that could do automation tasks instead of complex ruby scripts or whatever else is used by existing configuration management software.

Basher was fun to write, and for what it does it works pretty well. As part of writing it I also wrote this style guide to show 1. how I write bash and 2. how bash can be safe and predictable if written carefully.

This guide will try to be as objective as possible, providing reasoning for why certain decisions were made. For choices that are purely aesthetic (and may not be universally agreeable) they will exist in the `Aesthetics` section below.

Aesthetics

Tabs / Spaces

tabs.

Columns

not to exceed 80.

Semicolons

You don't use semicolons on the command line (I hope), don't use them in scripts.

```
# wrong
name='dave';
echo "hello $name";

#right
name='dave'
echo "hello $name"
```

The exception to this rule is outlined in the `Block Statements` section below. Namely, semicolons should be used for control statements like `if` or `while`.

Functions

Don't use the `function` keyword. All variables created in a function should be made local.

```
# wrong
function foo {
    i=foo # this is now global, wrong depending on intent
}

# right
foo() {
    local i=foo # this is local, preferred
}
```

Block Statements

`then` should be on the same line as `if`, and `do` should be on the same line as `while`.

```
# wrong
if true
then
    ...
fi

# also wrong, though admittedly looks kinda cool
true && {
    ...
}

# right
if true; then
    ...
fi
```

Spacing

No more than 2 consecutive newline characters (ie. no more than 1 blank line in a row)

Comments

No explicit style guide for comments. Don't change someones comments for aesthetic reasons unless you are rewriting or updating them.

Bashisms

This style guide is for bash. This means when given the choice, always prefer bash builtins or keywords instead of external commands or `sh(1)` syntax.

`test(1)`

Use `[[ ... ]]` for conditional testing, not `[ .. ]` or `test ...`

```
# wrong
test -d /etc

# also wrong
[ -d /etc ]

# correct
[[ -d /etc ]]
```

See http://mywiki.wooledge.org/BashFAQ/031 for more information

Sequences

Use bash builtins for generating sequences

```
n=10

# wrong
for f in $(seq 1 5); do
    ...
done

# wrong
for f in $(seq 1 "$n"); do
    ...
done

# right
for f in {1..5}; do
    ...
done

# right
for ((i = 0; i < n; i++)); do
    ...
done
```

Command Substitution

Use `$(...)` for command substitution.

```
foo=`date`  # wrong
foo=$(date) # right
```

Math / Integer Manipulation

Use `((...))` and `$((...))`.

```
a=5
b=4

# wrong
if [[ $a -gt $b ]]; then
    ...
fi

# right
if ((a > b)); then
    ...
fi
```

Do **not** use the `let` command.

Parameter Expansion

Always prefer parameter expansion (http://mywiki.wooledge.org/BashGuide/Parameters#Parameter_Expansion) over external commands like `echo`, `sed`, `awk`, etc.

```
name='bahamas10'

# wrong
prog=$(basename "$0")
nonumbers=$(echo "$name" | sed -e 's/[0-9]//g')

# right
prog=${0##*/}
nonumbers=${name//[0-9]/}
```

Listing Files

Do not parse ls(1) (http://mywiki.wooledge.org/ParsingLs), instead use bash builtin functions to loop files

```
# very wrong, potentially unsafe
for f in $(ls); do
    ...
done

# right
for f in *; do
    ...
done
```

Determining path of the executable (`__dirname`)

Simply stated, you can't know this for sure. If you are trying to find out the full path of the executing program, you should rethink your software design.

See http://mywiki.wooledge.org/BashFAQ/028 for more information

For a case study on `__dirname` in multiple languages see my blog post

http://daveeddy.com/2015/04/13/dirname-case-study-for-bash-and-node/

Arrays and lists

Use bash arrays instead of a string separated by spaces (or newlines, tabs, etc.) whenever possible

```
# wrong
modules='json httpserver jshint'
for module in $modules; do
    npm install -g "$module"
done

# right
modules=(json httpserver jshint)
for module in "${modules[@]}"; do
    npm install -g "$module"
done
```

Of course, in this example it may be better expressed as:

```
npm install -g "${modules[@]}"
```

… if the command supports multiple arguments, and you are not interested in catching individual failures.

read builtin

Use the bash `read` builtin whenever possible to avoid forking external commands

Example

```
fqdn='computer1.daveeddy.com'

IFS=. read -r hostname domain tld <<< "$fqdn"
echo "$hostname is in $domain.$tld"
# => "computer1 is in daveeddy.com"
```

External Commands

GNU userland tools

The whole world doesn't run on GNU or on Linux; avoid GNU specific options when forking external commands like `awk`, `sed`, `grep`, etc. to be as portable as possible.

When writing bash and using all the powerful tools and builtins bash gives you, you'll find it rare that you need to fork external commands to do simple string manipulation.

UUOC (http://www.smallo.ruhr.de/award.html)

Don't use `cat(1)` when you don't need it. If programs support reading from stdin, pass the data in using bash redirection.

```
# wrong
cat file | grep foo

# right
grep foo < file

# also right
grep foo file
```

Prefer using a command line tools builtin method of reading a file instead of passing in stdin. This is where we make the inference that, if a program says it can read a file passed by name, it's probably more performant to do that.

Style

Quoting

Use double quotes for strings that require variable expansion or command substitution interpolation, and single quotes for all others.

```
# right
foo='Hello World'
bar="You are $USER"

# wrong
foo="hello world"

# possibly wrong, depending on intent
bar='You are $USER'
```

All variables that will undergo word-splitting *must* be quoted (1). If no splitting will happen, the variable may remain unquoted.

```
foo='hello world'

if [[ -n $foo ]]; then   # no quotes needed:
                         # [[ ... ]] won't word-split variable expansions

    echo "$foo"          # quotes needed
fi

bar=$foo  # no quotes needed - variable assignment doesn't word-split
```

1. The only exception to this rule is if the code or bash controls the variable for the duration of its lifetime. For instance, basher (https://github.com/bahamas10/basher) has code like:

```
printf_date_supported=false
if printf '%()T' &>/dev/null; then
    printf_date_supported=true
fi

if $printf_date_supported; then
    ...
fi
```

Even though `$printf_date_supported` undergoes word-splitting in the `if` statement in that example, quotes are not used because the contents of that variable are controlled explicitly by the programmer and not taken from a user or command.

Also, variables like `$$` , `$?` , `$#` , etc. don't required quotes because they will never contain spaces, tabs, or newlines.

When in doubt however, quote all expansions (http://mywiki.wooledge.org/Quotes).

Variable Declaration

Avoid uppercase variable names unless there's a good reason to use them. Don't use `let` or `readonly` to create variables. `declare` should *only* be used for associative arrays. `local` should *always* be used in functions.

```
# wrong
declare -i foo=5
let foo++
readonly bar='something'
FOOBAR=baz

# right
i=5
((i++))
bar='something'
foobar=baz
```

shebang

Bash is not always located at `/bin/bash` , so use this line:

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

Unless you have a reason to use something else.

Error Checking

`cd` , for example, doesn't always work. Make sure to check for any possible errors for `cd` (or commands like it) and exit or break if they are present.

```
# wrong
cd /some/path # this could fail
rm file       # if cd fails where am I? what am I deleting?

# right
cd /some/path || exit
rm file
```

`set -e`

Don't set `errexit`. Like in C, sometimes you want an error, or you expect something to fail, and that doesn't necessarily mean you want the program to exit.

This is a contreversial opinion that I have on the surface, but the link below will show situations where `set -e` can do more harm than good because of its implications.

http://mywiki.wooledge.org/BashFAQ/105

`eval`

Never.

Common Mistakes

Using {} instead of quotes.

Using `${f}` is potentially different than `"$f"` because of how word-splitting is performed. For example.

```
for f in '1 space' '2  spaces' '3   spaces'; do
    echo ${f}
done
```

yields

```
1 space
2 spaces
3 spaces
```

Notice that it loses the amount of spaces. This is due to the fact that the variable is expanded and undergoes word-splitting because it is unquoted. This loop results in the 3 following commands being executed:

```
echo 1 space
echo 2  spaces
echo 3   spaces
```

The extra spaces are effectively ignored here and only 2 arguments are passed to the `echo` command in all 3 invocations.

If the variable was quoted instead:

```
for f in '1 space' '2  spaces' '3   spaces'; do
    echo "$f"
done
```

yields

```
1 space
2  spaces
3   spaces
```

The variable `$f` is expanded but doesn't get split at all by bash, so it is passed as a single string (with spaces) to the `echo` command in all 3 invocations.

Note that, for the most part `$f` is the same as `${f}` and `"$f"` is the same as `"${f}"` . The curly braces should only be used to ensure the variable name is expanded properly. For example:

```
$ echo "$HOME is $USERs home directory"
/home/dave is  home directory
$ echo "$HOME is ${USER}s home directory"
/home/dave is daves home directory
```

The braces in this example were the difference of `$USER` vs `$USERs` being expanded.

Abusing for-loops when while would work better

`for` loops are great for iteration over arguments, or arrays. Newline separated data is best left to a `while read -r ...` loop.

```
users=$(awk -F: '{print $1}' /etc/passwd)
for user in $users; do
    echo "user is $user"
done
```

This example reads the entire `/etc/passwd` file to extract the usernames into a variable separated by newlines. The `for` loop is then used to iterate over each entry.

This approach has a lot of issues if used on other files with data that may contain spaces or tabs.

1. This reads *all* usernames into memory, instead of processing them in a streaming fashion.
2. If the first field of that file contained spaces or tabs, the for loop would break on that as well as newlines
3. This only works *because* `$users` is unquoted in the `for` loop - if variable expansion only works for your purposes while unquoted this is a good sign that something isn't implemented correctly.

To rewrite this:

```
while IFS=: read -r user _; do
    echo "$user is user"
done < /etc/passwd
```

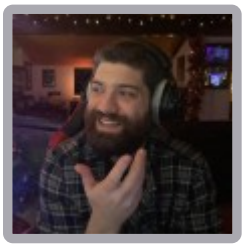
This will read the file in a streaming fashion, not pulling it all into memory, and will break on colons extracting the first field and discarding (storing as the variable `_`) the rest - using nothing but bash builtin commands.

Extra

- http://mywiki.wooledge.org/BashPitfalls

License

MIT License



(http://illumos.org/)

site last updated: 2022-03-16 02:37:22 EDT