Stuck in the 70’s?: Sanitize shell script from the disco era

Stuck in the 70's

Stuck in the 70’s?: Sanitize shell script from the disco era

Why bash?

As part of my work I encounter many shell scripts which are poorly written but are essential for the infrastructure to function properly.

Most of the time the stake holders aren’t sufficiently trained to use a higher level language such as **`python`**, **`ruby`** or **`go`**.

By introducing modern **`bash`** features, the scripts can be maintained by all stake holders without creating high hurdles for entry.

Many times during my professional career I encountered the magical script written by a former employee who left years ago. Said script was written in an obscure language that was all the hype at the time but its star has faded since. No one in the team has the time or is willing to learn the language, port it to a more common language or they are simply too afraid of touching the code.

I’m sure many have experienced similar situations. This presentation is an introduction to modern **`bash`** script concepts to help sanitize old shell scripts in need of a bit polish.

Target audience

If you often encounter shell or scripts written in **`bash`** and want to learn a few tricks to make your and your co-worker’s life easier this presentation is for you.

A basic understanding of programming in general and shell scripting in particular is assumed.

Errors, Shmerrors

Shell scripts are amazingly forgiving when it comes to bad coding practices. Virtually every variable is globally available and can be used without being previously declared.

Further, fatal errors in other languages result in the termination of the script, not so in a classic shell script. Even after a problem the script jugs along as if everything is dandy.

set -o errexit

The scripts I usually encounter do not exit on error. With this option set the script is terminated when an error occurs.

Script without **`errexit`** *option*

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

false
echo 'Still alive?' # Still alive?; exit rc 0
```

Script with **`errexit`** *option*

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

set -o errexit

false               # exit rc 1
echo 'Still alive?'
```

To prevent exit on error, append **`|| :`** after the command.

Prevent exit on error

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

set -o errexit

false || :
echo 'Still alive?' # Still alive?; exit rc 0
```

set -o nounset

Imagine a scenario where you remove a directory within a script where the last part of the path is provided by a variable. For one reason or another said variable is not defined and instead of the target directory you remove a lot more than you bargained for.

`rm -rf /home/jdoe/${temp_dir}`

But don't dispair with **`set -o nounset`** this can be prevented.

Script without **`nounset`** *option*

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

rm -rf /home/jdoe/${temp_dir} # rm -rf /home/jdoe/
```

Script with **`nounset`** *option*

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

set -o nounset

rm -rf /home/jdoe/${temp_dir} # bash: temp_dir: unbound variable; exit rc 1
```

set -o pipefail

When working with pipes per default only the right most command in the chain is considered and checked for errors.

With **`pipefail`** each command within the pipe construct is evaluted. This is especially useful for **`awk`** commands which generally return and exit status of **`0`**.

Script without **`pipefail`** *option*

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

find /fake-dir | awk '{print $1}' # exit rc 0
```

Script with **`pipefail`** *option*

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

set -o pipefail

find /fake-dir | awk '{print $1}' # exit rc 1
```

Summary

Always use the option triplet **`errexit`**, **`nounset`** and **`pipefail`** in your scripts!

When refactoring an existing script add them first to see how far down the rabbit whole the journey goes. If the script runs without a hitch the code might not be as bad as assumed.

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

set -o errexit
set -o nounset
set -o pipefail

[...]
```

To disable the option for a certain block or command use **+o** instead of **-o**.

```
[...]

# switch off exit on error
set +o errexit
false
# switch it back on
set -o errexit

[...]
```

The options for **errexit** and **nounset** can be written as **set -e** or **set -u** respectively. But I prefer writing them out to help a novice user to better understand these settings.

I do declare!

Variables in shell scripts are typeless, quite a few popular scripting languages do the same and are loosely typed. With **`declare`** certain attributes can be defined to make a script more predictable and sometimes cut down on conversion logic.

Integer

Ensure the variable only holds integers.

```
declare -i int=1

int=string    # bash: string: unbound variable

int=2         # declare -i int="2"
```

Readonly

Define a variable as a constant.

```
declare -r readonly=set-in-stone

readonly=change   # bash: readonly: readonly variable
```

Lowercase

The content of the variable is converted to lowercase.

```
declare -l lower=LOWER

echo ${lower}    # lower
```

Requires bash >= 4.0

Uppercase

The content of the variable is converted to uppercase.

```
declare -u upper=upper

echo ${upper}    # UPPER
```

Arrays

The content of the variable is an array.

```
declare -a array=( 1 2 3 )

typeset -p array  # declare -a array=([0]="1" [1]="2" [2]="3" )

array=foo

typeset -p array  # declare -a array=([0]="foo" [1]="2" [2]="3")
```

Associative Arrays

The content of the variable is an associative array.

```
declare -A hash=( [a]=foo [b]=bar )

typeset -p hash # declare -A hash=([b]="bar" [a]="foo" )

hash=c

typeset -p hash # declare -A hash=([0]="c" [b]="bar" [a]="foo" )

hash+=([c]=foo)

typeset -p hash # declare -A hash=([c]="foo" [b]="bar" [a]="foo" )
```

Requires bash >= 4.0

Summary

There are few other switches to **`declare`** and the certainly can be combined. To create a readonly array use **`declare -ra`** for instance.

Using **`declare`** helps narrowing the scope of the values a variable can hold. This makes the script more predictable!

Operation plus equal

With Bash 3.1 the new assignment operator **+=** was introduced. It makes the previously cumbersome task of appending or adding content to an already populated variable a lot easier and DRYer. The following examples show the classical way and the new improved way.

Integer incrementation

There is a few ways incrementing integers in bash but the focus is on the **+=** assignment operator.

Classical way

```
counter=0
while true; do
  counter=`expr ${counter} + 1`   # declare -- counter="1"
  # [...]
done
```

*With **+=** operator*

```
declare -i counter=0
while true; do
  counter+=1                      # declare -i counter="1"
  # [...]
done
```

Without declaring the variable as an integer the behavior is not as expected.

```
counter=0
while true; do
  counter+=1                      # declare -- counter="01"
  # [...]
done
```

Append to string

The **+=** assignment operator acts differently when a string is encountered.

Classical way

```
string="foo"
string="${string}bar"  # declare -- string="foobar"
```

*With **+=** operator*

```
declare -- string=foo
string+=bar            # declare -- string="foobar"
```

Push to array

Using the **+=** assignment operator with a list or array pushes a new value into an array.

Classical list

```
list="foo"
list="${list} bar" # declare -- list="foo bar"
```

*With **+=** operator*

```
declare -a list=( foo )
list+=( bar )      # declare -a list=(foo bar)
```

Summary

The **+=** operator helps with the assignment of existing and brings a bit more comfort to the previously wordy and often times ugly process of appending or incrementing numbers.

A **-=** assignment operator does not exist!

Substitution Revolution

Some may know the classical parameter expansion from Bourne shell such as **`${parameter:N,N}`** or **`${parameter:-default}`** or the ones from Korn shell **`${parameter%pattern}`** or **`${parameter#pattern}`**. But bash has a few of its own. We look at three of them.

Modify case

There are two basic modes, the first is to convert the case of the first letter. The second is to convert the whole string.

```
declare -- to_lower="ALL CAPS"
echo ${to_lower,}               # aLL CAPS
echo ${to_lower,,}              # all caps

declare -- to_upper="all lower"
echo ${to_upper^}               # All lower
echo ${to_upper^^}              # ALL LOWER
```

Requires bash >= 4.0

There is also the option of only matching a pattern to adjust the case.

```
declare -- fruits="banana apple pear"
echo ${fruits^a}                # banana apple pear
echo ${fruits^[bp]}             # Banana apple pear
echo ${fruits^^a}               # bAnAnA Apple peAr
echo ${fruits^^[ae]}            # bAnAnA ApplE pEAr
```

Requires bash >= 4.0

Real world example

```
if [ `echo ${string1} | tr "[A-Z]" "[a-z]"` = "${string2}" ]; then
  # [...]
fi
```

With parameter expansion

```
if [[ ${string1,,} == ${string2} ]]; then
  # [...]
fi
```

Substitution

This one really gets me! I almost never see **`${parameter/pattern/}`** substitutions in the wild. And is has been around for ages.

Given it is not as powerful as **sed** but for most use cases it is more than sufficient.

```
declare -- string="lagoon racoon"
echo ${string/oo/u}             # lagun racoon
echo ${string//oo/u}            # lagun racun
```

Real world example

```
string="lagoon racoon"
echo ${string} | sed 's/oo/u/g' # lagun racun
```

With parameter expansion

```
declare -- string="lagoon racoon"
echo ${string//oo/u}             # lagun racun
```

The script below loops 1000 times and does a string substitution with **sed** and then via the **bash** builtin substitution. The result is quite stunning. The builtin is is factors faster.

Example script **letter-substitution.sh**

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

set -o errexit
set -o nounset
set -o pipefail

declare -r TITLE="\nSubstitute 'o' for 'u' in string with '%s' a 1000 times\n"
declare -r STRING="lagoon racoon spoon loon"

function sed-loop() {
  printf "${TITLE}" 'sed'
  for i in {0..1000}; do
    sed 's/o/u/g' <<< ${STRING} &>/dev/null
  done
}

function builtin-loop() {
  printf "${TITLE}" '${var//o/u}'
  for i in {0..1000}; do
    echo ${STRING//o/u} &>/dev/null
  done
}

time sed-loop
time builtin-loop
```

Results

```
$ bash letter-substitution.sh

Substitute 'o' for 'u' in string with 'sed' a 1000 times

real    0m1.613s (1)
user    0m1.008s
sys     0m0.669s

Substitute 'o' for 'u' in string with '${var//o/u}' a 1000 times

real    0m0.020s (2)
user    0m0.020s
sys     0m0.000s
```

(1) **sed** takes about 8 times longer to complete the job

(2) Builtin is blazingly fast as it does not spawns a subshell for every substitution.

Summary

Parameter expansion in shell scripts can be a bit cryptic at first but it is definitely a lot faster then to run a sub shell for every little change made to a small string.

There are many more expansions available but the examples here are bash specific and won't work in the predecessor shells such as Bourne or Korn shell.

(hip hip)

Arrays have been around in bash for quite some time but the encounters in the real world are few and far between.

Combined with the parameter expansion they become a powerful asset DRYing up a WET script.

Here a few examples.

Modify case

Combining bash arrays with the upper case parameter expansion is mighty powerful. Let's see how it's done.

```
declare -a fruit=( banana apple pear )
echo ${fruit[@]^}               # Banana Apple Pear
echo ${fruit[@]^^}              # BANANA APPLE PEAR
echo ${fruit[@]^a}              # banana Apple pear
echo ${fruit[@]^[bp]}           # Banana apple Pear
echo ${fruit[@]^^a}             # bAnAnA Apple peAr
echo ${fruit[@]^^[ae]}          # bAnAnA ApplE pEAr
```

Requires bash >= 4.0

Substitution

Applying substitution to each element of an array can also be done. Here a few examples.

General substitutions

```
declare -a array=( lagoon racoon )
echo ${array[@]/o/u}          # laguon racuon
echo ${array[@]//o/u}         # laguun racuun
```

Prefix substitutions

```
declare -a sshopts=( BatchMode=yes User=foobar )
echo ${sshopts[@]/#/-o }      # -o BatchMode=yes -o User=foobar
```

Append suffix

```
declare -a fruits=( banana apple pear )
echo ${fruits[@]/%/,}         # banana, apple, pear,
```

Remove suffix

```
declare -a fruits=( bananas apples pears )
echo ${fruits[@]/%s/}         # banana apple pear
```

Summary

Parameter expansion combined with bash arrays allows to make volatile changes to a list of values at the time of echoing. No **`sed`**, **`tr`** and **`awk`** constructs and excessive looping are required.

Perfect condition

Conditional statements in classical shell script have a few shortcomings. For backward compatibility bash understands the single brackets **[…]** but are a couple more ways to enclose conditional statements.

Let's dive in.

Double square brackets

Double square brackets are more forgiving when it comes to dealing with empty strings in a comparison.

Classical shell script condition

```
foo=
[ ${foo} = foo ]      # bash: [: =: unary operator expected
```

Effectively the above conditional statement is expanded as **[= foo]** because the variable **foo** is empty. There are two common ways to prevent the error:

Classical shell script condition workaround

```
foo=
[ "${foo}" = "foo" ]  # exit rc 1
[ x${foo} = xfoo ]    # exit rc 1
```

Korn shell introduced the more robusts **[[** which does not suffer the same limitations.

Korn shell style double square brackets

```
declare -- foo=
[[ ${foo} == foo ]]   # exit rc 1
```

Korn was initially released in 1983. **bash** and **zsh** among other shells adapted the double square bracket style conditionals. Still to this day there I mostly encounter single square brackets all over the place. Hence the question "Are we stuck in the 70's?".

And while the double square brackets are not POSIx compliant, does it really matter if you are writing a bash script?

Regex comparison

Bash has also a regex operator **=~** for matching strings. This often overlooked feature can save you lots of typing.

More importantly string comparisons within bash are more performant as no sub shell is required for the comparison.

Classical shell string match

```
string=foobar
if echo ${string} | grep -q '[Bb]ar'; then
  # [ ... ]
fi
```

Bash style regex

```
declare -- string=foobar
if [[ ${string} =~ [Bb]ar ]]; then
  # [ ... ]
fi
```

Summary

These are but a few small examples creating "perfect conditions" for your bash scripts. Although they are not POSIX compatible the fact that **`ksh`**, **`zsh`** and **`bash`** among other implement them makes them virtually portable.

Funky, funky, functions!

Function names are quite restrictive in many languages. For instance minuses **-** are not permitted or special characters can not be used.

While this is true for traditional shell scripts as well Bash has more relaxed rules.

Let's dive in!

Special characters

With the exception of a few reserved characters such as **$**, **|**, **(** and **{** among others pretty much everything else works.

```
function @() { echo '@'; }
@                                       # @
function /() { echo '/'; }
/                                       # /
function -() { echo '-'; }
-                                       # -
function :() { echo ':'; }
:                                       # :
function ü() { echo 'ü'; }
ü                                       # ü
```

This opens possibilities of creating prefixes or fake namespaces for functions. Pretty useful for sourced files and libraries.

Functions with faux namespaces

```
function funky() {
  # [ ... ]
}

function test::funky() {
  # [ ... ]
}

funky            # execute funky
test::funky      # test funky function
```

Function within functions

It it is even possible to place functions within functions. Unfortunately there they are still accessible globally.

There is a use case for this scenario which we get to later. But with the **`local`** keyword the scope can be limited.

```
function parent() {
  function @child() {
    # [ ... ]
  }
  @child
  # [ ... ]
  @child
}
```

Summary

Surprisingly bash functions can be creatively named and have fewer restrictions than most other scripting languages. Caution is advised when using special characters tho. I don't think many bash scripters use them.

What the printf?

The builtin **`printf`** function comes with a few differences compared to the ones found in the common scripting languages.

Here we have a look how we can use **`printf`** a log smarter!

Don't loop, printf!

The bash **`printf`** can be used instead of a loop for printing as it prints each additional argument not unlike a loop.

Print fruit loop :)

```
declare -a fruit=( banana apple pear )
for f in ${fruit[@]}; do echo ${f}; done    # banana\napple\npear
```

Printf it!

```
declare -a fruit=( banana apple pear )
printf "%s\n" "${fruit[@]}"                 # banana\napple\npear
```

Straight lines

Don't manually write out lines use **`printf`**!

```
printf "%0.1s" -{1..16}         # ----------------
```

Packing the above into a function and calling it **`-----`** can make the source code more readable but probably confuses the casual code reviewer.

Example script **`pretty-lines.sh`**

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

function -----() {
  printf "%0.1s" -{1..16} $'\n'
}

-----
echo "Title"
-----
```

Result

```
$ bash pretty-lines.sh
----------------
Title
----------------
```

sprintf anyone?

One can even emulate the **`sprintf`** function by using the **`-v`** switch.

```
printf -v line "%0.1s" -{1..16}
echo ${line}                    # ----------------
```

Requires bash >= 3.1

Escape plan!

With the **`%q`** placeholder **`printf`** provides a shell escaping routine.

```
printf "%q" 'cat foo | grep "foo bar"'
# cat\ foo\ \|\ grep\ \"foo\ bar\"
```

What does the clock say?

Use **`printf`** instead of the **`date`** command to convert an Unix epoch timestamp to a human readable format.

```
printf "%(%Y-%m-%d)T\n" 1515151515
2018-01-05
```

Requires bash >= 4.2

Summary

IMHO, one of most underused builtin in bash scripts is printf. It is very versatile and can help reduce code and complexity if used appropriately.

Real World Examples ™

A few examples on how to reduce clutter with examples taken from the Real World ™.

The power of read(ing)

Expensive parsing and splitting with awk

```
info="Joe:Doe:Sysadmin"
fname=`echo ${info} | awk -F : '{ print $1 }'`   # Joe
lname=`echo ${info} | awk -F : '{ print $2 }'`   # Doe
role=`echo ${info} | awk -F : '{ print $3 }'`    # Sysadmin
```

read with IFS

```
declare -- info="Joe:Doe:Sysadmin"
IFS=: read fname lname role <<< "${info}"
```

The same but read everything into an array.

read into array

```
declare -- info="Joe:Doe:Sysadmin"
IFS=: read -a person <<< "${info}"

typeset -p person   # declare -a person=([0]="Joe" [1]="Doe" [2]="Sysadmin")
```

Splitting field separated data into variables is usually down with **awk** in shell scripts. And **awk** is superb tools when working on large data sets. However simply splitting into shell variables is a performance killer!

Example script **`read-list.sh`**

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

set -o errexit
set -o nounset
set -o pipefail

declare -r TITLE="\nSplit string to variables with '%s' a 100 times\n"
declare -r LIST="root:T0pS3cr3t!:0:0:root:/root:/bin/bash"

function awk-loop() {
  printf "${TITLE}" 'awk -F : ...'
  for i in {0..100}; do
    login=$(echo ${LIST} | awk -F : '{ print $1 }')
    pw=$(echo ${LIST} | awk -F : '{ print $2 }')
    uid=$(echo ${LIST} | awk -F : '{ print $3 }')
    gid=$(echo ${LIST} | awk -F : '{ print $4 }')
    gecos=$(echo ${LIST} | awk -F : '{ print $5 }')
    home=$(echo ${LIST} | awk -F : '{ print $6 }')
    shell=$(echo ${LIST} | awk -F : '{ print $7 }')
  done
}

function read-loop() {
  printf "${TITLE}" 'IFS=: read ...'
  for i in {0..100}; do
    IFS=: read login pw uid gid gecos home shell <<< "${LIST}"
  done
}

time awk-loop
time read-loop
```

```
$ bash read-list.sh

Split string to variables with 'awk -F : ...' a 100 times

real    0m1.162s (1)
user    0m1.086s
sys     0m0.293s

Split string to variables with 'IFS=: read ...' a 100 times

real    0m0.002s (2)
user    0m0.001s
sys     0m0.002s
```

(1) Splitting with **`awk`** is about 580 times slower! Although there are only 100 iterations **`awk`** is called 700 times.

(2) Barely noticeable the time used with the builtin.

May the case be with you

Too WET to maintain

```
if [ ${host} = jp-prod ]; then
  prod_host ${host}
elif [ ${host} = ch-prod ]; then
  prod_host ${host}
elif [ ${host} = jp-uat ]; then
  uat_host ${host}
# [ ... ]
else
  dev_host ${host}
fi
```

Better with case

```
case ${host} in
  *-prod) prod_host ${host};;
  *-uat)  uat_host ${host};;
  *)      dev_host ${host};;
esac
```

Poor man's Ansible

Execute a locally defined function on a remote machine without first copying the code.

```
function bootstrap() { config_host; hostname; }

ssh ch-dev.acme.com "$(declare -f bootstrap); bootstrap"
```

It is hard to imagine something concrete with the above example hence a small example to show how it works. In the below example the script is adding a new SSH public key to the target user's authorized_keys file. But only if the key is not already existing.

Example script **`rex-add_ssh_key.sh`**

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

function add_ssh_key() {
  set -o errexit
  set -o nounset
  set -o pipefail

  declare -r SSH_TYPE=${1}; shift;
  declare -r SSH_KEY=${1}; shift;
  declare -r SSH_COMMENT=${1}; shift;
  declare -r AUTH_KEYS=${HOME}/.ssh/authorized_keys

  function key_already_in_place() {
    grep -q ${SSH_KEY} ${AUTH_KEYS}
  }

  function add_key() {
    key_already_in_place && {
      echo "Key already exists in ${AUTH_KEYS}";
      return 0;
    }
    printf "%s %s %s\n" \
      ${SSH_TYPE} \
      ${SSH_KEY} \
      ${SSH_COMMENT} \
      >> ${AUTH_KEYS}
    echo "Key successfully added to ${AUTH_KEYS}"
  }
  add_key
}

ssh localhost \
  "$(declare -f add_ssh_key);" \
  "add_ssh_key ssh-ed25519 AAAAC3N..7mG testkey"
```

Results from running above script

```
$ bash rex-add_ssh_key.sh
Key succesfully added to /home/test/.ssh/authorized_keys (1)

$ bash rex-add_ssh_key.sh
Key already exists in /home/test/.ssh/authorized_keys (2)
```

(1) Running the first time the key is added

(2) Running the second time the key is already present and no action is taken.