Scripting with style

Fix Me! continue

These are some coding guidelines that helped me to read and understand my own code over the years. They also will help to produce code that will be a bit more robust than "if something breaks, I know how to fix it".

This is not a bible, of course. But I have seen so much ugly and terrible code (not only in shell) during all the years, that I'm 100% convinced there needs to be *some* code layout and style. No matter which one you use, use it throughout your code (at least don't change it within the same shellscript file); don't change your code layout with your mood.

Some good code layout helps you to read your own code after a while. And of course it helps others to read the code.

Indentation guidelines

Indentation is nothing that technically influences a script, it's only for us humans.

I'm used to seeing/using indentation of *two space characters* (though many may prefer 4 spaces, see below in the discussion section):

- it's easy and fast to type
- it's not a hard-tab that's displayed differently in different environments
- it's wide enough to give a visual break and small enough to not waste too much space on the line

Speaking of hard-tabs: Avoid them if possible. They only make trouble. I can imagine one case where they're useful: Indenting here-documents.

Breaking up lines

Whenever you need to break lines of long code, you should follow one of these two rules:

Indention using command width:

```
activate some_very_long_option \
         some_other_option
```

Indention using two spaces:

```
activate some_very_long_option \
  some_other_option
```

Personally, with some exceptions, I prefer the first form because it supports the visual impression of "these belong together".

Breaking compound commands

Compound commands form the structures that make a shell script different from a stupid enumeration of commands. Usually they contain a kind of "head" and a "body" that contains command lists. This type of compound command is relatively easy to indent.

I'm used to (not all points apply to all compound commands, just pick the basic idea):

- put the introducing keyword and the initial command list or parameters on one line ("head")
- put the "body-introducing" keyword on the same line
- the command list of the "body" on separate lines, indented by two spaces
- put the closing keyword on a separated line, indented like the initial introducing keyword

What?! Well, here again:

Symbolic

```
HEAD_KEYWORD parameters; BODY_BEGIN
  BODY_COMMANDS
BODY_END
```

if/then/elif/else

This construct is a bit special, because it has keywords (`elif` , `else`) "in the middle". The visually appealing way is to indent them like this:

```
if ...; then
  ...
elif ...; then
  ...
else
  ...
fi
```

for

```
for f in /etc/*; do
  ...
done
```

while/until

```
while [[ $answer != [YyNn] ]]; do
  ...
done
```

The case construct

The `case` construct might need a bit more discussion here, since its structure is a bit more complex.

In general, every new "layer" gets a new indentation level:

```
case $input in
  hello)
    echo "You said hello"
  ;;
  bye)
    echo "You said bye"
    if foo; then
      bar
    fi
  ;;
  *)
    echo "You said something weird..."
  ;;
esac
```

Some notes:

- if not 100% needed, the optional left parenthesis on the pattern is not used
- the patterns (`hello)`) and the corresponding action terminator (`;;`) are indented at the same level
- the action command lists are indented one more level (and continue to have their own indentation, if needed)
- though optional, the very last action terminator is given

Syntax and coding guidelines

Cryptic constructs

Cryptic constructs, we all know them, we all love them. If they are not 100% needed, avoid them, since nobody except you may be able to decipher them.

It's - just like in C - the middle ground between smart, efficient and readable.

If you need to use a cryptic construct, include a comment that explains what your "monster" does.

Variable names

Since all reserved variables are `UPPERCASE` , the safest way is to only use `lowercase` variable names. This is true for reading user input, loop counting variables, etc., … (in the example: `file`)

- prefer `lowercase` variables
- if you use `UPPERCASE` names, **do not use reserved variable names** (see SUS (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html#tag_(for an incomplete list)
- if you use `UPPERCASE` names, prepend the name with a unique prefix (`MY_` in the example below)

```
#!/bin/bash

# the prefix 'MY_'
MY_LOG_DIRECTORY=/var/adm/

for file in "$MY_LOG_DIRECTORY"/*; do
  echo "Found Logfile: $file"
done
```

Variable initialization

As in C, it's always a good idea to initialize your variables, though, the shell will initialize fresh variables itself (better: Unset variables will generally behave like variables containing a null string).

It's no problem to pass an **environment variable** to the script. If you blindly assume that all variables you use for the first time are **empty**, anybody can **inject** content into a variable by passing it via the environment.

The solution is simple and effective: **Initialize them**

```
my_input=""
my_array=()
my_number=0
```

If you do that for every variable you use, then you also have some in-code documentation for them.

Parameter expansion

Unless you are really sure what you're doing, **quote every parameter expansion**.

There are some cases where this isn't needed from a technical point of view, e.g.

- inside `[[ ... ]]` (other than the RHS of the `==` , `!=` , and `=~` operators)
- the parameter (`WORD`) in `case $WORD in ….`
- variable assignment: `VAR=$WORD`

But quoting these is never a mistake. If you quote every parameter expansion, you'll be safe.

If you need to parse a parameter as a list of words, you can't quote, of course, e.g.

```
list="one two three"

# you MUST NOT quote $list here
for word in $list; do
  ...
done
```

Function names

Function names should be all `lowercase` and meaningful. The function names should be human readable. A function named `f1` may be easy and quick to write down, but for debugging and especially for other people, it reveals nothing. Good names help document your code without using extra comments.

do not use command names for your functions. e.g. naming a script or function `test`, will collide with the UNIX `test` command.

Unless absolutely necessary, only use alphanumeric characters and the underscore for function names. `/bin/ls` is a valid function name in Bash, but is not a good idea.

Command substitution

As noted in the article about command substitution, you should use the `$( … )` form.

If portability is a concern, use the backquoted form `` ` … ` ``.

In any case, if other expansions and word splitting are not wanted, you should quote the command substitution!

Eval

Well, like Greg says: **"If eval is the answer, surely you are asking the wrong question."**

Avoid it, unless absolutely neccesary:

- `eval` can be your neckshot
- there are most likely other ways to achieve what you want
- if possible, re-think the way your script works, if it seems you can't avoid `eval` with your current method
- if you really, really, have to use it, then take care, and be sure about what you're doing

Basic structure

The basic structure of a script simply reads:

```
#!SHEBANG

CONFIGURATION_VARIABLES

FUNCTION_DEFINITIONS

MAIN_CODE
```

The shebang

If possible (I know it's not always possible!), use a shebang.

Be careful with `/bin/sh` : The argument that "on Linux `/bin/sh` is Bash" **is a lie** (and technically irrelevant)

The shebang serves two purposes for me:

- it specifies the interpreter to be used when the script file is called directly: If you code for Bash, specify `bash` !
- it documents the desired interpreter (so: use `bash` when you write a Bash-script, use `sh` when you write a general Bourne/POSIX script, …)

Configuration variables

I call variables that are meant to be changed by the user "configuration variables" here.

Make them easy to find (directly at the top of the script), give them meaningful names and maybe a short comment. As noted above, use `UPPERCASE` for them only when you're sure about what you're doing. `lowercase` will be the safest.

Function definitions

Unless there are reasons not to, all function definitions should be declared before the main script code runs. This gives a far better overview and ensures that all function names are known before they are used.

Since a function isn't parsed before it is executed, you usually don't have to ensure they're in a specific order.

The portable form of the function definition should be used, without the `function` keyword (here using the grouping compound command):

```
getargs() {
  ...
}
```

Speaking about the command grouping in function definitions using `{ …; }` : If you don't have a good reason to use another compound command directly, you should always use this one.

Behaviour and robustness

Fail early

Fail early, this sounds bad, but usually is good. Failing early means to error out as early as possible when checks indicate an error or unmet condition. Failing early means to error out **before** your script begins its work in a potentially broken state.

Availability of commands

If you use external commands that may not be present on the path, or not installed, check for their availability, then tell the user they're missing.

Example:

```
my_needed_commands="sed awk lsof who"

missing_counter=0
for needed_command in $my_needed_commands; do
  if ! hash "$needed_command" >/dev/null 2>&1; then
    printf "Command not found in PATH: %s\n" "$needed_command" >&2
    ((missing_counter++))
  fi
done

if ((missing_counter > 0)); then
  printf "Minimum %d commands are missing in PATH, aborting\n" "$miss
ing_counter" >&2
  exit 1
fi
```

Exit meaningfully

The exit code is your only way to directly communicate with the calling process without any special provisions.

If your script exits, provide a meaningful exit code. That minimally means:

- `exit 0` (zero) if everything is okay
- `exit 1` - in general non-zero - if there was an error

This, **and only this**, will enable the calling component to check the operation status of your script.

You know: **"One of the main causes of the fall of the Roman Empire was that, lacking zero, they had no way to indicate successful termination of their C programs."** – *Robert Firth*

Misc

Output and appearance

- if the script is interactive, if it works for you and if you think this is a nice feature, you can try to save the terminal content and restore it after execution
- output clean and understandable screen messages
- if applicable, you can use colors or specific prefixes to tag error and warning messages
    - make it easy for the user to identify those messages

- write normal output to `STDOUT` . write error, warning and diagnostic messages to `STDERR`
  - enables message filtering
  - keeps the script from mixing output data with diagnostic, or error messages
  - if the script gives syntax help (`-?` or `-h` or `–help` arguments), it should go to `STDOUT`
- if applicable, write error/diagnostic messages to a logfile
  - avoids screen clutter
  - messages are available for diagnostic use

Input

- never blindly assume anything. If you want the user to input a number, **check for numeric input, leading zeros**, etc. If you have specific format or content needs, **always validate the input!**

Tooling

- some of these guidelines, such as indentation, positioning of "body-introducing" keywords, and portable function declarations, can be enforced by shfmt (https://github.com/mvdan/sh)

Discussion

Jari Aalto (http://wiki.debian.org/JariAalto), 2010/11/09 18:33 ()

About then indentation amount: The industry standard is pretty much 4, which translates to *half-tab*.

(1) Try adjusting screen resolution to higher ones, do you still feel comfortabel with the indentation? (2) Try adjusting font size of your editor to smaller fonts to increase visible are and line count, is the indentation still clear? (3) try printing code fitted 4 pages in one sheet, do you still read the code indentation?

For all these point, 4 works, any less usually have problems when taken out of context of an individual developer. Thee *half-tab* on the other hand adapts without problems to variety of environments.

Lioba, 2013/11/29 19:07 ()

Fine Jari, you must work in a different industry I can not feel more disagree. It looks like you are talking about other different programming or scripting language. You must try to read and write a lot of scripts in the real combat sands of terminals before understand you are wrong but may be you will. Do you actually need to print 4 pages of bash scripting fitting one A4 then you have bigger problems than your indentation you have mistake your work choosing. Why are you as sure as

everybody works on kind of editors where you can "change the size of my font to a smaller one" or "adjust the screen resolution"? Are you use to work in terminals or terminal emulators? It looks like not and, if so, why are you ignoring font Types are usually monospaced just for a clearer reading?

Alastair Andrew, 2011/03/28 23:26 ()

What about the Bash 4 command_not_found_handle function? Implementing that would appear to be a much cleaner way of handling the situation where a command that is relied upon cannot be found.

Jan Schampera, 2011/03/29 05:19 (), 2011/03/29 05:29 ()

Several reasons I don't like `command_not_found_handle()` in this situation

- the basic idea is good, the actual way is an ugly hack IMHO (though the discussion about a better way I had with the Bash maintainer wasn't very successful)
- regarding the way it is implemented and how it really works (see below), it's a feature primarily designed for interactive use
- it would mean to fail when the command is to be used, not early in a pre-flight-check (bad code sanity, IMHO)
- it is difficult seeing the failed command(s):

```
$ command_not_found_handle() { echo "Not found: ''$@''"; }
$ echa foo | grep bar | grep baz             # ''echa'' - it doesn't output - it's swallowed by the pipe
$ echa foo | grep bar | greb baz             # now it outputs, only for ''greb''
Not found: ''greb baz''
```

  The main reason here is that the function is ran in the same environment as the command that was about to be executed (it's executed in the pipe instead of "echa") - something you may not want in all cases

Of course you can script around the third issue, using several `BASH_*` variables and correct filedescriptors and stuff, but you can catch it all in a small loop iterating over the needed commands and telling the user exactly which command is missing.

Martin Kealey, 2012/08/06 06:46 ()

Your suggested layout for compound commands, while in keeping with other programming languages (and indeed, commonly used for shell scripting because of that), rather misses the point about shell grammar.

Two connected points in particular: semicolon and end-of-line are both terminators, and the "head" portion is (in most cases) a list.

So it makes sense in at least some cases to put the interceding word on its own line.

Firstly, in a "while" loop, sometimes you want to do things *before* the test:

```
while
  do_this
  do_that
  test_something
do
  do_something_else
done
```

In extremis, the "body" may even be empty, providing the equivalent of the C-language "do … while" construct:

```
while
  do_something
  test_something
do :; done
```

(I happen to thing it would be a nice extension to allow the "do LIST" to be omitted entirely.)

Secondly, in a "for" loop, you might have long items which are clearer if aligned vertically:

```
for x in path/to/one/thing \
         path/to/another/thing \
         path/to/yet/another/thing \
         a/path/to/somewhere else
do
  do_something $x
done
```

In these cases I think that having the "do" on a separate line makes it clearer where the "head" list ends and the "body" list begins.

The "head" in an "if" statement is similarly a list, and again it makes sense to visually separate a compound "head" from a "body" (although this only makes a difference if separated by something other than semicolons, or if used in a pipeline):

```
if X=$( generate_list |
        grep -qs "$quick_filter" ) &&
   [[ $X = "$precise_thing_to_check" ]]
then
  echo FOUND "$X"
else
  echo NOT FOUND
fi
```

(and yes, most of the time I prefer 4-spaces rather than 2)

Jan Schampera, 2012/08/25 17:33 ()

I agree.

The text needs some tweaking. I know those constructs and use them myself. I just didn't have them in mind here. The text is far from complete, yes.

Christopher Barry, 2012/09/30 21:30 ()

RE: Indentation. Always a sticky wicket, and definitely a religious thing, but I have found that it's always better to avoid actual tab characters, instead configuring your editor to use spaces for tabs. Personally, I prefer 4-space tabs.

The reason I say this, is that when coding in my editor, I like to cut and paste functions into the shell and execute them as I refine the function. This workflow technique works well for me, you may have your own way. The point is, pasting in tabs can cause all kinds of weird errors, so I avoid them like the plague. I also use emacs with sh-mode, and I like how it works.

I think a good way to format your code is the same way that bash itself re-formats your code. Try this in a shell:

copy a fairly involved function from one of your scripts, and paste it into a shell to instantiate it. say the function is named 'my_func'. Now do the following (assuming tabs did not blow it up :):

$ declare -f my_func

If the output looks exactly like you coded it, you're good. If not, try to code the way bash itself outputs it. You'll notice there are no tab characters in the output.

My 2-cents.

Arild Jensen, 2013/08/01 21:58 ()

The link http://www.opensolaris.org/os/project/shell/shellstyle/ (http://www.opensolaris.org/os/project/shell/shellstyle/) is dead.

Alister, 2013/10/19 02:52 ()

> Avoid if, unless absolutely neccesary…

Should this say "avoid it…"?

Hemal Pandya, 2014/08/29 18:49 ()

It would be useful if you explain why `eval` is bad and the circumstances in which it is useful.

Kevin Briggs, 2016/09/14 13:17 ()

It seems like the 'eval is bad' argument is just a blindly repeated mantra that really popped up just because its *dangerous*. As with all things, I think if you know what you're doing, it's *good*. It just depends on the perspective. From the perspective of writing portable, reusable code that may be picked up, edited, and used by bash novices, maybe it is bad. From the perspective of using it by a bash expert to do some complex indirection, maybe it's good. Either way, it's a unique tool that can potentially do things no other bash tool can.

alex, 2017/07/19 19:52 ()

i was wondering the same thing :)

Zibri (http://twitter.com/Zibri), 2016/12/15 01:16 ()

In bash I do this:

```
  source <(echo "Zibri () {";cat script_to_be_reindented.sh; echo
"}")
  declare -f Zibri| cut -c 5-|head --lines=-1|tail --lines=+3
```

this eliminates comments and reindents the script "bash way". it will not work if the script contains HEREDOCS but if you do this:

```
  source <(echo "Zibri () {";cat script_to_be_reindented.sh; echo
"}")
  declare -f Zibri|head --lines=-1|tail --lines=+3
```

it will work with any script but the whole script will be indented by 4 spaces. feel free to modify but cite my name in your script and post it! :D

David, 2017/04/23 21:35 ()

Like this page, it's a very good reference.

On the function naming part, I have some additional conventions that I use.

For library type functions the form I use is this:

```
f_array.GetIndexOf
f_array.Reverse
```

For other script-specific function names I have been using a convention that is perhaps useful to others. I try to separate what my functions do with:

```
f_get_IpOfDomain     returns content.
f_set_NumberOfCalls  changes a global variable.
f_do_SaveSession     makes changes to a system or host, more than
writing to /tmp.
```

And Im considering the following naming convention for menus with infinite while-loops:

```
f_show_MainMenu
f_show_SelectionMenu
```