Shell Style Guide

Revision 1.26

Paul Armstrong
Too many more to mention

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: ▽ . You may toggle all summaries with the big arrow button:

▽ Toggle all summaries

Table of Contents

Shell Files and Interpreter Invocation	File Extensions SUID/SGID
Environment	STDOUT vs STDERR
Comments	File Header Function Comments Implementation Comments TODO Comments
Formatting	Indentation Line Length and Long Strings Pipelines Loops Case statement Variable expansion Quoting
Features and Bugs	Command Substitution Test, [and [[Testing Strings Wildcard Expansion of Filenames Eval Pipes to While
Naming Conventions	Function Names Variable Names Constants and Environment Variable Names Source Filenames Read-only Variables Use Local Variables Function Location main
Calling Commands	Checking Return Values Builtin Commands vs. External Commands
Conclusion	

Background

Which Shell to Use

link ▽ `Bash` is the only shell scripting language permitted for executables.

Executables must start with `#!/bin/bash` and a minimum number of flags. Use `set` to set shell options so that calling your script as `bash <script_name>` does not break its functionality.

Restricting all executable shell scripts to bash gives us a consistent shell language that's installed on all our machines.

The only exception to this is where you're forced to by whatever you're coding for. One example of this is Solaris SVR4 packages which require plain Bourne shell for any scripts.

When to use Shell

link ▽ Shell should only be used for small utilities or simple wrapper scripts.

While shell scripting isn't a development language, it is used for writing various utility scripts throughout Google. This style guide is more a recognition of its use rather than a suggestion that it be used for widespread deployment.

Some guidelines:

- If you're mostly calling other utilities and are doing relatively little data manipulation, shell is an acceptable choice for the task.
- If performance matters, use something other than shell.
- If you find you need to use arrays for anything more than assignment of `${PIPESTATUS}`, you should use Python.
- If you are writing a script that is more than 100 lines long, you should probably be writing it in Python instead. Bear in mind that scripts grow. Rewrite your script in another language early to avoid a time-consuming rewrite at a later date.

Shell Files and Interpreter Invocation

File Extensions

link ▽ Executables should have no extension (strongly preferred) or a `.sh` extension. Libraries must have a `.sh` extension and should not be executable.

It is not necessary to know what language a program is written in when executing it and shell doesn't require an extension so we prefer not to use one for executables.

However, for libraries it's important to know what language it is and sometimes there's a need to have similar libraries in different languages. This allows library files with identical purposes but different languages to be identically named except for the language-specific suffix.

SUID/SGID

link ▽ SUID and SGID are *forbidden* on shell scripts.

There are too many security issues with shell that make it nearly impossible to secure sufficiently to allow SUID/SGID. While bash does make it difficult to run SUID, it's still possible on some platforms which is why we're being explicit about banning it.

Use `sudo` to provide elevated access if you need it.

Environment

STDOUT vs STDERR

link ▽ All error messages should go to `STDERR`.

This makes it easier to separate normal status from actual issues.

A function to print out error messages along with other status information is recommended.

```
err() {
  echo "[$(date +'%Y-%m-%dT%H:%M:%S%z')]: $@" >&2
}

if ! do_something; then
```

```
    err "Unable to do_something"
    exit "${E_DID_NOTHING}"
  fi
```

Comments

File Header

link ▽ Start each file with a description of its contents.

Every file must have a top-level comment including a brief overview of its contents. A copyright notice and author information are optional.

Example:

```
#!/bin/bash
#
# Perform hot backups of Oracle databases.
```

Function Comments

link ▽ Any function that is not both obvious and short must be commented. Any function in a library must be commented regardless of length or complexity.

It should be possible for someone else to learn how to use your program or to use a function in your library by reading the comments (and self-help, if provided) without reading the code.

All function comments should contain:

- Description of the function
- Global variables used and modified
- Arguments taken
- Returned values other than the default exit status of the last command run

Example:

```
#!/bin/bash
#
# Perform hot backups of Oracle databases.

export PATH='/usr/xpg4/bin:/usr/bin:/opt/csw/bin:/opt/goog/bin'

#######################################
# Cleanup files from the backup dir
# Globals:
#   BACKUP_DIR
#   ORACLE_SID
# Arguments:
#   None
# Returns:
#   None
#######################################
cleanup() {
  ...
}
```

Implementation Comments

link ▽ Comment tricky, non-obvious, interesting or important parts of your code.

This follows general Google coding comment practice. Don't comment everything. If there's a complex algorithm or you're doing something out of the ordinary, put a short comment in.

TODO Comments

link ▽ Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect.

This matches the convention in the C++ Guide.

TODOs should include the string TODO in all caps, followed by your username in parentheses. A colon is optional. It's preferable to put a bug/ticket number next to the TODO item as well.

Examples:

```
# TODO(mrmonkey): Handle the unlikely edge cases (bug ####)
```

Formatting

While you should follow the style that's already there for files that you're modifying, the following are required for any new code.

Indentation

link ▽ Indent 2 spaces. No tabs.

Use blank lines between blocks to improve readability. Indentation is two spaces. Whatever you do, don't use tabs. For existing files, stay faithful to the existing indentation.

Line Length and Long Strings

link ▽ Maximum line length is 80 characters.

If you have to write strings that are longer than 80 characters, this should be done with a here document or an embedded newline if possible. Literal strings that have to be longer than 80 chars and can't sensibly be split are ok, but it's strongly preferred to find a way to make it shorter.

```
# DO use 'here document's
cat <<END;
I am an exceptionally long
string.
END

# Embedded newlines are ok too
long_string="I am an exceptionally
  long string."
```

Pipelines

link ▽ Pipelines should be split one per line if they don't all fit on one line.

If a pipeline all fits on one line, it should be on one line.

If not, it should be split at one pipe segment per line with the pipe on the newline and a 2 space indent for the next section of the pipe. This applies to a chain of commands combined using '|' as well as to logical compounds using '||' and '&&'.

```
# All fits on one line
command1 | command2

# Long commands
command1 \
  | command2 \
  | command3 \
    | command4
```

Loops

link

▽ Put `; do` and `; then` on the same line as the `while`, `for` or `if`.

Loops in shell are a bit different, but we follow the same principles as with braces when declaring functions. That is: `; then` and `; do` should be on the same line as the if/for/while. `else` should be on its own line and closing statements should be on their own line vertically aligned with the opening statement.

Example:

```
for dir in ${dirs_to_cleanup}; do
  if [[ -d "${dir}/${ORACLE_SID}" ]]; then
    log_date "Cleaning up old files in ${dir}/${ORACLE_SID}"
    rm "${dir}/${ORACLE_SID}/"*
    if [[ "$?" -ne 0 ]]; then
      error_message
    fi
  else
    mkdir -p "${dir}/${ORACLE_SID}"
    if [[ "$?" -ne 0 ]]; then
      error_message
    fi
  fi
done
```

Case statement

link

▽

- Indent alternatives by 2 spaces.
- A one-line alternative needs a space after the close parenthesis of the pattern and before the `;;`.
- Long or multi-command alternatives should be split over multiple lines with the pattern, actions, and `;;` on separate lines.

The matching expressions are indented one level from the 'case' and 'esac'. Multiline actions are indented another level. In general, there is no need to quote match expressions. Pattern expressions should not be preceded by an open parenthesis. Avoid the `;&` and `;;&` notations.

```
case "${expression}" in
  a)
    variable="..."
    some_command "${variable}" "${other_expr}" ...
    ;;
  absolute)
    actions="relative"
    another_command "${actions}" "${other_expr}" ...
    ;;
  *)
    error "Unexpected expression '${expression}'"
    ;;
esac
```

Simple commands may be put on the same line as the pattern *and* `;;` as long as the expression remains readable. This is often appropriate for single-letter option processing. When the actions don't fit on a single line, put the pattern on a line on its own, then the actions, then `;;` also on a line of its own. When on the same line as the actions, use a space after the close parenthesis of the pattern and another before the `;;`.

```
verbose='false'
aflag=''
bflag=''
```

```
files=''
while getopts 'abf:v' flag; do
  case "${flag}" in
    a) aflag='true' ;;
    b) bflag='true' ;;
    f) files="${OPTARG}" ;;
    v) verbose='true' ;;
    *) error "Unexpected option ${flag}" ;;
  esac
done
```

Variable expansion

link ▽ In order of precedence: Stay consistent with what you find; quote your variables; prefer "${var}" over "$var", but see details.

These are meant to be guidelines, as the topic seems too controversial for a mandatory regulation.
They are listed in order of precedence.

1. Stay consistent with what you find for existing code.
2. Quote variables, see Quoting section below.
3. Don't brace-quote single character shell specials / positional parameters, unless strictly necessary or avoiding deep confusion.
   Prefer brace-quoting all other variables.

```
# Section of recommended cases.

# Preferred style for 'special' variables:
echo "Positional: $1" "$5" "$3"
echo "Specials: !=$!, -=$-, _=$_. ?=$?, #=$# *=$* @=$@ \$=$$ ..."

# Braces necessary:
echo "many parameters: ${10}"

# Braces avoiding confusion:
# Output is "a0b0c0"
set -- a b c
echo "${1}0${2}0${3}0"

# Preferred style for other variables:
echo "PATH=${PATH}, PWD=${PWD}, mine=${some_var}"
while read f; do
  echo "file=${f}"
done < <(ls -l /tmp)

# Section of discouraged cases

# Unquoted vars, unbraced vars, brace-quoted single letter
# shell specials.
echo a=$avar "b=$bvar" "PID=${$}" "${1}"

# Confusing use: this is expanded as "${1}0${2}0${3}0",
# not "${10}${20}${30}
set -- a b c
echo "$10$20$30"
```

Quoting

link ▽
- Always quote strings containing variables, command substitutions, spaces or shell meta characters, unless careful unquoted expansion is required.
- Prefer quoting strings that are "words" (as opposed to command options or path names).
- Never quote *literal* integers.
- Be aware of the quoting rules for pattern matches in [[.
- Use "$@" unless you have a specific reason to use $*.

```
# 'Single' quotes indicate that no substitution is desired.
# "Double" quotes indicate that substitution is required/tolerated.

# Simple examples
# "quote command substitutions"
flag="$(some_command and its args "$@" 'quoted separately')"

# "quote variables"
echo "${flag}"

# "never quote literal integers"
value=32
# "quote command substitutions", even when you expect integers
number="$(generate_number)"

# "prefer quoting words", not compulsory
readonly USE_INTEGER='true'

# "quote shell meta characters"
echo 'Hello stranger, and well met. Earn lots of $$$'
echo "Process $$: Done making \$\$\$."

# "command options or path names"
# ($1 is assumed to contain a value here)
grep -li Hugo /dev/null "$1"

# Less simple examples
# "quote variables, unless proven false": ccs might be empty
git send-email --to "${reviewers}" ${ccs:+"--cc" "${ccs}"}

# Positional parameter precautions: $1 might be unset
# Single quotes leave regex as-is.
grep -cP '([Ss]pecial|\|?characters*)$' ${1:+"$1"}

# For passing on arguments,
# "$@" is right almost everytime, and
# $* is wrong almost everytime:
#
# * $* and $@ will split on spaces, clobbering up arguments
#   that contain spaces and dropping empty strings;
# * "$@" will retain arguments as-is, so no args
#   provided will result in no args being passed on;
#   This is in most cases what you want to use for passing
#   on arguments.
# * "$*" expands to one argument, with all args joined
#   by (usually) spaces,
#   so no args provided will result in one empty string
#   being passed on.
# (Consult 'man bash' for the nit-grits ;-)

set -- 1 "2 two" "3 three tres"; echo $# ; set -- "$*"; echo "$#, $@")
set -- 1 "2 two" "3 three tres"; echo $# ; set -- "$@"; echo "$#, $@")
```

Features and Bugs

Command Substitution

link ▽ Use `$(command)` instead of backticks.

Nested backticks require escaping the inner ones with `\`. The `$(command)` format doesn't change when nested and is easier to read.

Example:

```
# This is preferred:
var="$(command "$(command1)")"
```

```
# This is not:
var="`command \`command1\``"
```

Test, [and [[

link

▽ `[[ ... ]]` is preferred over `[`, `test` and `/usr/bin/[`.

`[[ ... ]]` reduces errors as no pathname expansion or word splitting takes place between `[[` and `]]` and `[[ ... ]]` allows for regular expression matching where `[ ... ]` does not.

```
# This ensures the string on the left is made up of characters in the
# alnum character class followed by the string name.
# Note that the RHS should not be quoted here.
# For the gory details, see
# E14 at https://tiswww.case.edu/php/chet/bash/FAQ
if [[ "filename" =~ ^[[:alnum:]]+name ]]; then
  echo "Match"
fi

# This matches the exact pattern "f*" (Does not match in this case)
if [[ "filename" == "f*" ]]; then
  echo "Match"
fi

# This gives a "too many arguments" error as f* is expanded to the
# contents of the current directory
if [ "filename" == f* ]; then
  echo "Match"
fi
```

Testing Strings

link

▽ Use quotes rather than filler characters where possible.

Bash is smart enough to deal with an empty string in a test. So, given that the code is much easier to read, use tests for empty/non-empty strings or empty strings rather than filler characters.

```
# Do this:
if [[ "${my_var}" = "some_string" ]]; then
  do_something
fi

# -z (string length is zero) and -n (string length is not zero) are
# preferred over testing for an empty string
if [[ -z "${my_var}" ]]; then
  do_something
fi

# This is OK (ensure quotes on the empty side), but not preferred:
if [[ "${my_var}" = "" ]]; then
  do_something
fi

# Not this:
if [[ "${my_var}X" = "some_stringX" ]]; then
  do_something
fi
```

To avoid confusion about what you're testing for, explicitly use `-z` or `-n`.

```
# Use this
if [[ -n "${my_var}" ]]; then
```

```
  do_something
fi

# Instead of this as errors can occur if ${my_var} expands to a test
# flag
if [[ "${my_var}" ]]; then
  do_something
fi
```

Wildcard Expansion of Filenames

link ▽ Use an explicit path when doing wildcard expansion of filenames.

As filenames can begin with a `-`, it's a lot safer to expand wildcards with `./*` instead of `*`.

```
# Here's the contents of the directory:
# -f  -r  somedir  somefile

# This deletes almost everything in the directory by force
psa@bilby$ rm -v *
removed directory: `somedir'
removed `somefile'

# As opposed to:
psa@bilby$ rm -v ./*
removed `./-f'
removed `./-r'
rm: cannot remove `./somedir': Is a directory
removed `./somefile'
```

Eval

link ▽ `eval` should be avoided.

Eval munges the input when used for assignment to variables and can set variables without making it possible to check what those variables were.

```
# What does this set?
# Did it succeed? In part or whole?
eval $(set_my_variables)

# What happens if one of the returned values has a space in it?
variable="$(eval some_function)"
```

Pipes to While

link ▽ Use process substitution or for loops in preference to piping to while. Variables modified in a while loop do not propagate to the parent because the loop's commands run in a subshell.

The implicit subshell in a pipe to while can make it difficult to track down bugs.

```
last_line='NULL'
your_command | while read line; do
  last_line="${line}"
done

# This will output 'NULL'
echo "${last_line}"
```

Use a for loop if you are confident that the input will not contain spaces or special characters (usually, this means not user input).

```
total=0
# Only do this if there are no spaces in return values.
for value in $(command); do
  total+="${value}"
done
```

Using process substitution allows redirecting output but puts the commands in an explicit subshell rather than the implicit subshell that bash creates for the while loop.

```
total=0
last_file=
while read count filename; do
  total+="${count}"
  last_file="${filename}"
done < <(your_command | uniq -c)

# This will output the second field of the last line of output from
# the command.
echo "Total = ${total}"
echo "Last one = ${last_file}"
```

Use while loops where it is not necessary to pass complex results to the parent shell - this is typically where some more complex "parsing" is required. Beware that simple examples are probably more easily done with a tool such as awk. This may also be useful where you specifically don't want to change the parent scope variables.

```
# Trivial implementation of awk expression:
#   awk '$3 == "nfs" { print $2 " maps to " $1 }' /proc/mounts
cat /proc/mounts | while read src dest type opts rest; do
  if [[ ${type} == "nfs" ]]; then
    echo "NFS ${dest} maps to ${src}"
  fi
done
```

Naming Conventions

Function Names

link

▽ Lower-case, with underscores to separate words. Separate libraries with `::`. Parentheses are required after the function name. The keyword `function` is optional, but must be used consistently throughout a project.

If you're writing single functions, use lowercase and separate words with underscore. If you're writing a package, separate package names with `::`. Braces must be on the same line as the function name (as with other languages at Google) and no space between the function name and the parenthesis.

```
# Single function
my_func() {
  ...
}

# Part of a package
mypackage::my_func() {
  ...
}
```

The `function` keyword is extraneous when "()" is present after the function name, but enhances quick identification of functions.

Variable Names

link ▽ As for function names.

Variables names for loops should be similarly named for any variable you're looping through.

```
for zone in ${zones}; do
  something_with "${zone}"
done
```

Constants and Environment Variable Names

link ▽ All caps, separated with underscores, declared at the top of the file.

Constants and anything exported to the environment should be capitalized.

```
# Constant
readonly PATH_TO_FILES='/some/path'

# Both constant and environment
declare -xr ORACLE_SID='PROD'
```

Some things become constant at their first setting (for example, via getopts). Thus, it's OK to set a constant in getopts or based on a condition, but it should be made readonly immediately afterwards. Note that `declare` doesn't operate on global variables within functions, so `readonly` or `export` is recommended instead.

```
VERBOSE='false'
while getopts 'v' flag; do
  case "${flag}" in
    v) VERBOSE='true' ;;
  esac
done
readonly VERBOSE
```

Source Filenames

link ▽ Lowercase, with underscores to separate words if desired.

This is for consistency with other code styles in Google: `maketemplate` or `make_template` but not `make-template`.

Read-only Variables

link ▽ Use `readonly` or `declare -r` to ensure they're read only.

As globals are widely used in shell, it's important to catch errors when working with them. When you declare a variable that is meant to be read-only, make this explicit.

```
zip_version="$(dpkg --status zip | grep Version: | cut -d ' ' -f 2)"
if [[ -z "${zip_version}" ]]; then
  error_message
else
  readonly zip_version
fi
```

Use Local Variables

link ▽ Declare function-specific variables with `local`. Declaration and assignment should be on different lines.

Ensure that local variables are only seen inside a function and its children by using `local` when declaring them. This avoids polluting the global name space and inadvertently setting variables that may have significance outside the function.

Declaration and assignment must be separate statements when the assignment value is provided by a command substitution; as the 'local' builtin does not propagate the exit code from the command substitution.

```
my_func2() {
  local name="$1"

  # Separate lines for declaration and assignment:
  local my_var
  my_var="$(my_func)" || return

  # DO NOT do this: $? contains the exit code of 'local', not my_func
  local my_var="$(my_func)"
  [[ $? -eq 0 ]] || return

  ...
}
```

Function Location

link ▽ Put all functions together in the file just below constants. Don't hide executable code between functions.

If you've got functions, put them all together near the top of the file. Only includes, `set` statements and setting constants may be done before declaring functions.

Don't hide executable code between functions. Doing so makes the code difficult to follow and results in nasty surprises when debugging.

main

link ▽ A function called `main` is required for scripts long enough to contain at least one other function.

In order to easily find the start of the program, put the main program in a function called `main` as the bottom most function. This provides consistency with the rest of the code base as well as allowing you to define more variables as `local` (which can't be done if the main code is not a function). The last non-comment line in the file should be a call to `main`:

```
main "$@"
```

Obviously, for short scripts where it's just a linear flow, `main` is overkill and so is not required.

Calling Commands

Checking Return Values

link ▽ Always check return values and give informative return values.

For unpiped commands, use `$?` or check directly via an `if` statement to keep it simple.

Example:

```
if ! mv "${file_list}" "${dest_dir}/" ; then
  echo "Unable to move ${file_list} to ${dest_dir}" >&2
  exit "${E_BAD_MOVE}"
fi
```

```
# Or
mv "${file_list}" "${dest_dir}/"
if [[ "$?" -ne 0 ]]; then
  echo "Unable to move ${file_list} to ${dest_dir}" >&2
  exit "${E_BAD_MOVE}"
fi
```

Bash also has the `PIPESTATUS` variable that allows checking of the return code from all parts of a pipe. If it's only necessary to check success or failure of the whole pipe, then the following is acceptable:

```
tar -cf - ./* | ( cd "${dir}" && tar -xf - )
if [[ "${PIPESTATUS[0]}" -ne 0 || "${PIPESTATUS[1]}" -ne 0 ]]; then
  echo "Unable to tar files to ${dir}" >&2
fi
```

However, as `PIPESTATUS` will be overwritten as soon as you do any other command, if you need to act differently on errors based on where it happened in the pipe, you'll need to assign `PIPESTATUS` to another variable immediately after running the command (don't forget that `[` is a command and will wipe out `PIPESTATUS`).

```
tar -cf - ./* | ( cd "${DIR}" && tar -xf - )
return_codes=(${PIPESTATUS[*]})
if [[ "${return_codes[0]}" -ne 0 ]]; then
  do_something
fi
if [[ "${return_codes[1]}" -ne 0 ]]; then
  do_something_else
fi
```

Builtin Commands vs. External Commands

link ▽ Given the choice between invoking a shell builtin and invoking a separate process, choose the builtin.

We prefer the use of builtins such as the *Parameter Expansion* functions in `bash(1)` as it's more robust and portable (especially when compared to things like sed).

Example:

```
# Prefer this:
addition=$((${X} + ${Y}))
substitution="${string/#foo/bar}"

# Instead of this:
addition="$(expr ${X} + ${Y})"
substitution="$(echo "${string}" | sed -e 's/^foo/bar/')"
```

Conclusion

Use common sense and *BE CONSISTENT*.

Please take a few minutes to read the Parting Words section at the bottom of the C++ Guide.

Revision 1.26