lh-style Documentation

Release 1.0.0

Luc Hermitte

Mar 08, 2021

Contents

1 Introduction **3**

2 Naming policy **5**
2.1 `:NameConvert policy` 5
2.2 `:[range]ConvertNames/pattern/policy/[flags]` 5
2.3 Options to tune the naming policy 5

3 Code formatting **7**
3.1 Style families 7
3.1.1 Families already implemented 7
3.1.2 `:UseStyle style-family=value [-buffer] [-ft[=ft]] [-prio=prio]` 8
3.1.3 `.editorconfig` 9
3.1.4 `.clang-format` 9
3.1.5 Extending the families 9
3.2 Low-level style configuration 10
3.2.1 `:AddStyle key [-buffer] [-ft[=ft]] [-prio=prio] Replacement` . . 10

4 API **13**
4.1 Formatting API 13
4.2 Naming API 13

5 Installation **15**

6 Indices and tables **17**

Contents:

CHAPTER 1

Introduction

lh-style is a vim script library that defines vim functions and commands that permit to specify stylistic preferences like naming conventions, bracket formatting, etc.

In itself the only feature end-users can directly exploit is name converting based on the style name (`snake_case`, `UpperCamelCase`...) like Abolish plugin does, or on a given identifier kind (*function*, *type*, *class*, *attribute*...). Check *:NameConvert policy* and *:[range]ConvertNames/pattern/policy/[flags]* – sorry I wasn't inspired.

The main, and unique, feature this plugin offers is core code-style functionalities that other plugins can exploit. Typical client plugins would be code generating plugins: wizards/snippet/abbreviation plugins, and refactoring plugins.

The style can be tuned through options. The options are meant to be tuned by end-users, and indirectly used by plugin maintainers. See *API section* to see how you could exploit these options from your plugins.

Snippets from lh-cpp and mu-template, and refactorings from vim-refactor exploit the options offered by lh-style for specifying code style.

> Note: The library has been extracted from lh-dev v2.x.x. in order to remove dependencies to lh-tags and other plugins from template/snippet expander plugins like mu-template. Yet, I've decided to reset the version counter to 1.0.0.

CHAPTER 2

Naming policy

Thanks to lh-style, we can define how the names of functions, classes, constants, attributes, etc. shall be written: in `UpperCamelCase`, in `lowerCamelCase`, in `snake_case`, or in `Any_otherStyle`).

This information can then be retrieved by plugins through the *Naming API*.

This information can also be used from *`:NameConvert`* and *`:ConvertNames`* commands.

NB: both commands support command-line auto-completion on naming policy names.

2.1 `:NameConvert` *`policy`*

`:NameConvert` converts the identifier under the cursor to one of the following naming policies:

- naming styles: `upper_camel_case`/`UpperCamelCase`, `lower_camel_case`/`lowerCamelCase`, `underscore`/`snake`, `UPPER_CASE`/`SCREAMING_SNAKE_CASE`, `variable`,
- identifier kinds: `getter`, `setter`, `local`, `global`, `member`, `static`, `constant`, `param` (the exact conversion process can be tuned thanks to the *following options*).

2.2 `:[range]ConvertNames/`*`pattern`*`/`*`policy`*`/[`*`flags`*`]`

`ConvertNames` transforms according to the *`policy`* all names that match the *`pattern`* – it applies `:NameConvert` on text matched by `:substitute`.

See :h :s_flags regarding possible *`flags`*.

2.3 Options to tune the naming policy

Naming conventions can be defined to:

- Control prefix and suffix on:

  - variables (main name)
  - global and local variables
  - member and static variables
  - (formal) parameters
  - constants
  - getters and setters
  - types
- Control the case policy (`snake_case`, `UpperCamelCase`, `lowerCamelCase`, `SCREAMING_SNAKE_CASE`) on functions (and thus on setters and getters too) and types.

It is done, respectively, with the following options:

- regarding prefix and suffix:
  - `(bpg):[{ft}_]naming_strip_re` and `(bpg):[{ft}_]naming_strip_subst`,
  - `(bpg):[{ft}_]naming_global_re`, `(bpg):[{ft}_]naming_global_subst`, `(bpg):[{ft}_]naming_local_re`, and `(bpg):[{ft}_]naming_local_subst`,
  - `(bpg):[{ft}_]naming_member_re`, `(bpg):[{ft}_]naming_member_subst`, `(bpg):[{ft}_]naming_static_re`, and `(bpg):[{ft}_]naming_static_subst`,
  - `(bpg):[{ft}_]naming_param_re`, and `(bpg):[{ft}_]naming_param_subst`,
  - `(bpg):[{ft}_]naming_constant_re`, and `(bpg):[{ft}_]naming_constant_subst`,
  - `(bpg):[{ft}_]naming_get_re`, `(bpg):[{ft}_]naming_get_subst`, `(bpg):[{ft}_]naming_set_re`, and `(bpg):[{ft}_]naming_set_subst`
  - `(bpg):[{ft}_]naming_type_re`, and `(bpg):[{ft}_]naming_type_subst`,
- regarding case:
  - `(bpg):[{ft}_]naming_function`
  - `(bpg):[{ft}_]naming_type`

Once in the *main name* form, the `..._re` regex options match the *main name* while the `..._subst` replacement text is applied instead.

You can find examples for these options in mu-template template used by BuildToolsWrapper's `:BTW new_project` command.

CHAPTER 3

Code formatting

Some projects will want to have open curly-brackets on new lines (see Allman indenting style), other will prefer to have the open bracket on the same line as the function/control-statement/... (see K&R indenting style, Java coding style...). We can also choose whether one space shall be inserted before opening braces, and so on.

Of course we could apply reformatting tools (like **`clang-format`**) on demand, but then we'd need to identify a set of different tools dedicated to different languages. The day code formatting is handled by Language Server Protocol, we would have access to a simple and extensible solution. In the mean time, here is lh-style.

lh-style doesn't do any replacement by itself on snippets or abbreviations. It is expected to be used by snippet plugins or from abbreviation definitions. So far, only mu-template and lh-cpp exploit this feature.

Different people will need to do different things:

- Plugin maintainers will use the *dedicated API* to reformat on-the-fly the code they generate.
- End-users will specify the coding style used on their project(s):
  - either by specifying a set of independent styles on different topics (*families*) (`:UseStyle`, `.editorconfig`, `.clang-format`),
  - or by being extremely precise (`:AddStyle`).

3.1 Style families

3.1.1 Families already implemented

At this time, the following style families are implemented:

- EditorConfig styles:
  - `curly_bracket_next_line` = [`yes`, `no`] // `true, false, 0, 1`.
  - `indent_brace_style` = [`0tbs`, `1tbs`, `allman`, `bsd_knf`, `gnu`, `horstmann`, `java`, `K&R`, `linux_kernel`, `lisp`, `none`, `pico`, `ratliff`, `stroustrup`, `whitesmiths`]
  - `spaces_around_brackets` = [`inside`, `outside`, `both`, `none`]

- Clang-Format styles:
  - `breakbeforebraces` = [`allman`, `attach`, `gnu`, `linux`, `none`, `stroustrup`]
  - `spacesbeforetrailingcomments` = [`\n`, positive number of spaces]
  - `spacesbeforeparens` = [`none`, `never`, `always`, `control-statements`]
  - `spacesinemptyparentheses` = [`yes`, `no`] // `true, false, 0, 1`.
  - `spacesinparentheses` = [`yes`, `no`] // `true, false, 0, 1`.

Styles *can easilly be added* in `{&rtp}/autoload/lh/style/`.

If you want more precise control, without family management, you can use *`:AddStyle`* instead.

3.1.2 `:UseStyle` *`style-family=value`* `[-buffer] [-ft[=`*`ft`*`]] [-prio=`*`prio`*`]`

Given *style families*, `:UseStyle` can be used to specify which particular style shall be used when generating code.

For instance,

```
:UseStyle breakbeforebraces=Allman -ft=c
:UseStyle spacesbeforeparens=control-statements -ft=c
:UseStyle spacesinparentheses=no
```

will tune snippets/abbreviations to follow Allman indenting style, and to add a space before control-statement parentheses, and to never insert a space inside parentheses.

Note: Some families are incompatible with other families. It happens quickly when we mix overlapping families from different origins.

`:UseStyle` options

- *`style-family=value`* specifies, given a style family, what choice has been made.

  If you don't remember them all, don't worry, `:UseStyle` supports command-line completion.

  Setting the style to `none` unsets the whole family for the related buffers/filetype. Giving a new *`value`*, overrides the style for the related buffers/filetype.
- `-buffer` defines this association only for the current buffer. This option is meant to be used with plugins like local_vimrc.
- `-ft[=`*`ft`*`]` defines this association only for the specified filetype. When *`ft`* is not specified, it applies only to the current filetype. This option is meant to be used in .vimrc, in the global zone of filetype-plugins or possibly in local_vimrcs (when combined with `-buffer`).
- `-prio=`*`prio`* Sets a priority that'll be used to determine which key is matching the text to enhance. By default all styles have a priority of 1. The typical application is to have template expander ignore single curly brackets.

Note: Local configuration (with `-buffer`) have the priority over filetype specialized configuration (with `-ft`).

3.1.3 .editorconfig

lh-style registers a hook to editorconfig-vim in order to extract style choices expressed in any `.editorconfig` file that applies.

The syntax would be:

```
[*]
indent_brace_style=Allman
```

In every buffer where EditorConfig applies its settings, it will be translated into:

```
:UseStyle -b indent_brace_style=allman
```

3.1.4 .clang-format

The idea is the same: to detect automatically a `.clang-format` configuration file in project root directory and apply the styles supported by lh-style.

Warning: At this time, this feature isn't implemented yet.

3.1.5 Extending the families

New style families can be defined (and even contributed back – as soon as I write the contributing guide...). The following procedure has to be respected:

1. Create a new autoload plugin named *rtp*/autoload/lh/style/*family-name*.vim
2. Define the following (required) functions:
   - lh#style#*family-name*#_known_list() which will be used by command-line completion
   - lh#style#*family-name*#use(*styles*, *value* [, *options*]) which defines the chosen style.

     The typical content of this function is the following:

```
function! lh#style#{family-name}#use(styles, value, ...) abort
  let input_options = get(a:, 1, {})
  let [options, local_global, prio, ft] = lh#style#_prepare_options_for_
↪add_style(input_options)

  " I usually use a `lh#style#{family-name}#__new()` function for this␣
↪purpose.
  let s:crt_style = lh#style#define_group('some.unique.family.id', name,␣
↪local_global, ft)

  " Then we dispatch the a:value option to decide how the text should be␣
↪displayed
  if     a:value =~? value_pattern1
    call s:crt_style.add(regex1, repl1, prio)
  elseif a:value =~? value_pattern2
    call s:crt_style.add(regex2, repl2, prio)
  else
```

(continues on next page)

(continued from previous page)

```
    call s:crt_style.add(regex3, repl3, prio)
  endif
  return 1
endfunction
```

Note:

- It'll be best to also define the other functions I have in all my autoload plugins in order to simplify logging and debugging.
- I highly recommand you take the time to write some unit tests – yeah, I know, I haven't written them for all possible cases supported by lh-style.

Todo:

- Describe `!cursorhere!`, `!mark!` and `lh#marker#txt()`
- Describe negative pattern
- Describe how priorities applies
- Describe other ways to dispatch
- Describe `none()`

3.2 Low-level style configuration

Historically, there wasn't any way to group style configurations as *`:UseStyle`*. permits. We add to define everything manually, and switching from one complex configuration to another was tedious.

While using *`:UseStyle`*. is now the preferred method, we can still use the low level method.

3.2.1 `:AddStyle` *`key`* `[-buffer] [-ft[=`*`ft`*`]] [-prio=`*`prio`*`]` *`Replacement`*

- *`key`* is a regex that will get replaced automatically (by plugins supporting this API)
- *`replacement`* is what will be inserted in place of *`key`*
- `-buffer` defines this association only for the current buffer. This option is meant to be used with plugins like local_vimrc.
- `-ft[=`*`ft`*`]` defines this association only for the specified filetype. When *`ft`* is not specified, it applies only to the current filetype. This option is meant to be used in .vimrc, in the global zone of filetype-plugins or possibly in local_vimrcs (when combined with `-buffer`).
- `-prio=`*`prio`* Sets a priority that'll be used to determine which key is matching the text to enhance. By default all styles have a priority of 1. The typical application is to have template expander ignore single curly brackets.

Note: Local configuration (with `-buffer`) have the priority over filetype specialized configuration (with `-ft`).

(Deprecated) `:AddStyle` Examples:

```
" # Space before open bracket in C & al {{{2
" A little space before all C constructs in C and child languages
" NB: the spaces isn't put before all open brackets
AddStyle if(     -ft=c   if\ (
AddStyle while(  -ft=c   while\ (
AddStyle for(    -ft=c   for\ (
AddStyle switch( -ft=c   switch\ (
AddStyle catch(  -ft=cpp catch\ (

" # Ignore style in comments after curly brackets {{{2
AddStyle {\ *// -ft=c &
AddStyle }\ *// -ft=c &

" # Multiple C++ namespaces on same line {{{2
AddStyle {\ *namespace -ft=cpp &
AddStyle }\ *} -ft=cpp &

" # Doxygen {{{2
" Doxygen Groups
AddStyle @{  -ft=c @{
AddStyle @}  -ft=c @}

" Doxygen Formulas
AddStyle \\f{ -ft=c \\\\f{
AddStyle \\f} -ft=c \\\\f}

" # Default style in C & al: Stroustrup/K&R {{{2
AddStyle {  -ft=c -prio=10 {\n
AddStyle }; -ft=c -prio=10 \n};\n
AddStyle }  -ft=c -prio=10 \n}

" # Inhibated style in C & al: Allman, Whitesmiths, Pico {{{2
" AddStyle {  -ft=c -prio=10 \n{\n
" AddStyle }; -ft=c -prio=10 \n};\n
" AddStyle }  -ft=c -prio=10 \n}\n

" # Ignore curly-brackets on single lines {{{2
AddStyle ^\ *{\ *$ -ft=c &
AddStyle ^\ *}\ *$ -ft=c &

" # Handle specifically empty pairs of curly-brackets {{{2
" On its own line
" -> Leave it be
AddStyle ^\ *{}\ *$ -ft=c &
" -> Split it
" AddStyle ^\ *{}\ *$ -ft=c {\n}

" Mixed
" -> Split it
" AddStyle {} -ft=c -prio=5 {\n}
" -> On the next line (unsplit)
AddStyle {} -ft=c -prio=5 \n{}
" -> On the next line (split)
" AddStyle {} -ft=c -prio=5 \n{\n}
```

(continues on next page)

(continued from previous page)

```
" # Java style {{{2
" Force Java style in Java
AddStyle { -ft=java -prio=10 {\n
AddStyle } -ft=java -prio=10 \n}
```

When you wish to adopt Allman coding style, in `$project_root/_vimrc_local.vim`

```
AddStyle { -b -ft=c -prio=10 \n{\n
AddStyle } -b -ft=c -prio=10 \n}
```

CHAPTER 4

API

This part is just a draft for the moment.

4.1 Formatting API

TBC

- `lh#style#clear()`
- `lh#style#get()`
- `lh#style#get_groups()`
- `lh#style#apply()`
- `lh#style#apply_these()`
- `lh#style#reinject_cached_ignored_matches()`
- `lh#style#ignore()`
- `lh#style#just_ignore_this()`
- `lh#style#surround()`
- `lh#style#use()`
- `lh#style#define_group()`

4.2 Naming API

TBC

- `lh#naming#variable()`
- `lh#naming#getter()`

- `lh#naming#setter()`
- `lh#naming#ref_getter()`
- `lh#naming#proxy_getter()`
- `lh#naming#global()`
- `lh#naming#local()`
- `lh#naming#member()`
- `lh#naming#static()`
- `lh#naming#constant()`
- `lh#naming#param()`
- `lh#naming#type()`
- `lh#naming#to_lower_camel_case()`
- `lh#naming#to_upper_camel_case()`
- `lh#naming#to_underscore()`

CHAPTER 5

Installation

- Requirements:
  - Vim 7.+,
  - lh-vim-lib (v5.3.1),
  - editorconfig-vim (optional).
- Install with vim-addon-manager any plugin that requires lh-style should be enough.
- With vim-addon-manager, install `lh-style` (this is the preferred method because of the dependencies).

```
ActivateAddons lh-style
" will also install editorconfig-vim
```

- vim-flavor (which also supports dependencies)

```
flavor 'LucHermitte/lh-style'
" will also install editorconfig-vim
```

- Vundle/NeoBundle:

```
Bundle 'LucHermitte/lh-vim-lib'
Bundle 'LucHermitte/lh-style'
" Optional
Bundle 'editorconfig/editorconfig-vim'
```

- Clone from the git repositories, and update your `'runtimepath'`

```
git clone git@github.com:LucHermitte/lh-vim-lib.git
git clone git@github.com:LucHermitte/lh-style.git
# Optional
git clone git@github.com:editorconfig/editorconfig-vim'
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search