Home

Style Guides

This is a collection of style guides for a wide selection of different programming languages. Each guide also provides a brief description of the language. More comparisions of the programming languages can be found in the Lexici repo.

C

C

C++

C++

STUFF

Introduction

Target Readership

Aims

Non-Aims

Enforcement

The Structure of this Document

Philosophy

Express Ideas Directly in Code

Write in ISO Standard C++

Express Intent

Ideally, a Program Should be Statically Type Safe

Prefer Compiler-Time Checking to Run-Time Checking

What Cannot be Checked at Compile Time Should be Checked at Run Time

Catch Run-Time Errors Early

Don't Leak Any Resources

Don't Waste Time or Space

Prefer Immutable Data to Mutable Data

Encapsulate Messy Constructs, Rather Than Spreading Through the Code

Use Supporting Tools as Appropriate

Use Support Libraries as Appropriate

Interfaces

Make Interfaces Explicit

Avoid Non-Const Global Variables

Avoid Singletons

Make Interfaces Precisely and Strongly Typed

State Preconditions

Prefer Expects for Expressing Preconditions

State Postconditions

Prefer Ensures for Expressing Postconditions

If an Interface is a Template, Document its Parameters Using Concepts

Use Exceptions to Signal a Failure to Preform a Required Task

Never Transfer Ownership by a Raw Pointer or Refernece

Declare a Pointer that Must not be Null as not_null

Do not Pass an Array as a Single Pointer

Avoid Complex Initialization of Global Objects

Keep the Number of Function Arguments Low

Avoid Adjacent Unrelated Parameters of the Same Type

Prefer Abstract Classes as Interfaces to Calss Hierarchies

If You Want a Cross-COmpiler ABI, Use a C-Stype Subset

For Stable Library ABI, Consider the Pimpl Idiom

Encapsulate Rule Violations

Functions

“Package” Meaningful Operations as Carefully Named Funcitons

A Function Should Preform a Single Logical Operation

Keep Functions Short and Simple

If a Function May Have to be Evaluated at Compile Time, Declare it Constexpr

If a Function is Very Small and Time-Critical, Declare it Inline

If Your Function May Not Throw, Declare it Noexcept

For General Use, Take T* or T& Arguments Rather Than Smart Pointers

Prefer Pure Functions

Unused Parameters Should be Unnamed

Classes and Class Hierarchies

Enumerations

Resource Management

Expressions and Statements

Preformance

Concurrency

Error Handling

Constants and Immutability

Templates and Generic Programming

C-Stype Programming

Source Files

The Standard Library

Architectural Ideas

Non-Rules and Myths

Profiles

Guideline Support Library

Naming and Layout

CommonLisp

Common Lisp

The Common Lisp language was developed as a standardized and improved successor of Maclisp. Byt the early 1980s several groups wer ealready at work on diverse successors to MacLisp. COmmon lisp sought to unify, standardise, and extend the features of these MaxLisp dialects. Common Lisp is not an implementation but rather a language spcifications. Common Lisp is a general-purpose, multi-paradigm programming lanugage. It supports a combinations of procedural, functions, and object-oriented programming paradigms. As a dynamic programming language, it facilitates evolutionary and incremental software development, wither iterative compilation into efficient run-tim programs. This incremental development is often done interactivly without interrupting the running application.

Meta-Guide

Must, Should, May, or Not

Each guideline's level of importance is indicated by use fo the following keywords and phrases.

MUST

This, or the terms "REQUIRED" or "SHALL", means that the guideline is an absolute requirement. You must ask permission to violate a MUST

MUST NOT

This phrase, or the phrase "SHALL NOT", means that the guideline is an absolute prohibition. You must ask permisson to violate a MUST NOT.

SHOULD

This word, or the adjective "RECOMMENDED", means that there may exist valid reasons in particular circumstances to ignore the demands of the guideline, but the full implications must be understood and carefully weighted before choosing a different course. You must ask forgiveness for violating a SHOULD.

SHOULD NOT

This phrase, or the phrase "NOT RECOMMENDED", means that there may exist valid rasons in particular circumstances to ignore thr prohibitions of this guideline, but the full implications should be understood and carefully weighted before choosing a different course. You must ask forgiveness for violating a SHOULD NOT.

MAY

This word, or the adjective "OPTIONAL", means that an item is truly optional.

Permission and Forgiveness

There are cases wher transgression of some of these rules is useful or even necessary. In some cases, you must seek permission or obtain forgiveness form the proper people.

Permission comes from the owners of your project.

Forgiveness is requested in a comment near the point of guideline violation, and is granted by your code reviewer. The original comment should be signed by you, the reviewer should add a signed approval to the comment at review time.

Conventions

You MUST follow conventsion. They are not optional.

Some of these guidelines are motivated by universal priciples of good programming. Some guidelines are motivated by technical peculiarities of COmmon Lisp. Some guidelines were once motivated by a technical reason, but the guideline remained after the reason subsided. Some guidelines, such those about comments and indentation, are based purely on convention, rather than on clear technical merit. Whatever the case may be, you must still follow these guidelines, aswell as other conventional guidelines that have not been fomalized in this document.

You MUST follow conventaions. THey are important for readability. When conventions are followed by default, violations of the convention are a signal that something notable is happening and deserves attention. When conventations are systematically violated, violations of the conventation are distractin noise that needs to be ignored.

Conventional guidelines *are* indoctrination. Their purpose is to make you follow the mores of the community, so you can more effectively cooperate with existing members. It is still useful to distinguish the parts that are technically motivated from the parts that are mear conventions, so you know when best to defy conventions for good effect, and when not to fall into the pitfalls that the conventions are there to help avoid.

General Guidelines

Principles

There are some basic principles for team software development that every developer must keep in mind. Whenever the detailed guidelines are inadequate, confusing or contradictory, refer back to these principles for guidance:

- Every developer's code must be easy for another developer to read, understand and modify – even if the first developer isn't around to explain it.

- Everybody's code should look the same.
- Be precise.
- Be concise.
- KISS – Keep It Simple, Stupid.
- Use the smallest hammer for the job.
- Use common sense.
- Keep related code together. Minimize the amount of jumping around someone has to do to understand an area of code.

Priorities

Architecture

Using Libraries

Open-Sourcing Code

Development Process

Formatting

Spelling and Abbreviations

Line Length

Indentation

File Header

Vertical Whitespace

Horizontal Whitespace

Documentation

Document Everything

Comment Semicolons

Grammar and Punctuation

Attention Required

Domain-Specific Languages

Naming

Symbol Guidelines

Denote Indent, not Content

Global Variables and Constants

Predicate Names

Omit Library Prefixes

Packages

Language Usage Guidelines

Mostly Functional Style

Recursion

Special Variables

Assignment

Assertions and Conditions

Type Checking

Identify, Equality and Comparisions

Iteration

I/O

Optimization

Avoid Allocation

Unsafe Operations

DYNAMIC-EXTENT

REDUCE vs APPLY

Avoid NCONC

Pitfalls

#'FUN vs 'FUN

Pathnames

SATISFIES

VimScript

VimScript

Vim Script is the scripting language built into Vim. Based on the ex editor language of the original vi editor, early versions of Vim added commands for control flow and function definitions. Since version 7, Vim script also supports more advanced data types such as lists and dictionaries and object-oriented programming. Built-in functions such as `map()` and `filter()` allow a basic form of functional programming, and Vim script has lambda since version 8.0. Vim script is mostly written in an imperative programming style.

Portability

It's hard to get vimscript right. Many commands depend upon the user;s settings. By following these guidelines, you can hope to make your scripts portable.

Strings

Prefer single quoted strings

Double quoted strings are semantically different in vimscript, and you probably don't want them (they break regexes).

Use double quoted string when you need an escape sequence (such as `"\n"`) or if you know it doesn't matter and you need to embed single quotes.

Matching Strings

Use the `=~#` or `=~?` operator families over the `=~` family.

The matching behavior depends upon the user's ignorecase and smartcase settings and on whether you compare them with the `=~`, `=~#` or `=~?` family of operators. Use the `=~#` and `=~?` operator families explicitly when comparing strings unless you explicitly need to honor the user's case sensitivit settings.

Regular Expressions

Prefer all regexes with `\m\C`.

In addition to the case sensitivity settings, regex behavior depends upon the user's nomagic setting. to make regexes act like nomagic and noignorecase are set, preend all regexes with `\m\C`.

You are welcome to use other magic levels (`\v`) and case sensitivities (`\c`) so long as they are intentional and explicit.

Dangerous Commands

Avoid commands with unintended side effects.

Avoid using `:s[ubstitute]` as it moves the cursor and prints error messages. Prefer functions (such as `search()`) better suited to scripts.

For many vim commands, functoins exist that do the same thing with fewer side effects. See `:help function()` for a list of build-in functions.

Fragile Commands

Avoid commands that rely on user settings.

Always use `normal!` instead of `normal`. The latter depends upon the user's key mappings and could do anything.

Avoid `:a[ubstitute]`, as its behavior depends upon a number of local settings.

The sam eapplies to other commands not listed here.

Catching Exceptions

Match error codes, not error text.

Error text may be locale dependant.

General Guidelines

Messaging

Message the user infrequently.

Loud scripts are annoying. Message the user only when:

- A long-running process has kicked off.
- An error has occured.

Type Checking

Use strict and explicit checks where possible.

Vimscript has unsafe, unintuitive behavior when dealing with some types. For instance, `0=='foo'` evaluates to true.

Use strict comparison operators where possible. When comparing against a string literal, use the `is#` operator. Otherwise, check `type()` explicitly.

Check variable types explicitly before using them. Use `type()` and throw your own errors.

Use `:unlet` for variables that may change types, particularly those assigned inside loops.

Python

Use sparingly.

Use python only when it provides critical functionality, for example when writing threaded code.

Other Languages

Use vimscript instead

Avoid using other scripting languages such as ruby and lua. We cannot guarantee that the end user's vim has been compiled with support for non-vimscript languages.

Boilerplate

Plugin boilerplate includes

- Plugin creation
- Error handling
- Dependency checking

Plugin Layout

Organize functionality into modular plugins

Group your functionality as a plugin, unified in one directory (or code repository) which shares your plugin's name (with a "vim-" prefix or ".vim" suffix if desired). It should be split into `plugin/`, `autoload/`, etc. subdirectories as necessary, and it should delcar metadata in the addon-info.json format.

Functions

In the `autoload/` directory, defined with `[!]` and `[abort]`.

Autoloading allows functions to be loaded on demand, which makes startuptime faster and enforces function namespacing.

Script-local functions are welcome, but should also live in `autoload/` and be called by autoloaded functions.

Non-library plugins should expose commands instead of functions. Command logic should be extracted into functions and autoloaded.

`[!]` allows developers to reload their functions without complaint.

`[abort]` forces the function to halt when it encounters an error.

Commands

In the `plugin/commands.vim` or under the `ftplugin/` directory, defined without `[!]`.

General commands go in `plugin/commands.vim`. Filetype-specific commands go in `ftplugin/`.

Excluding `[!]` prevents your plugin from silently clobbering existing commands. Command conflicts should be resolved by the user.

Autocommands

Place them in `plugin/autocmds.vim`, within augroups.

Place all autocommands in augroups.

The augroup name should be unique. It should either be, or be prefixed with, the plugin name.

Clear the augroup with `autocmd!` before defining new autocommands in the augroup. This makes your plugin re-entrable.

Mappings

Place them in `plugin/mappings.vim`, using a prefix.

All key mappings sould be defined in `plugin/mappings.vim`.

Partial mappings (see `:help using-<Plug>`) should be defined in `plugin/plugs.vim`.

Settings

Change settings locally.

Use `:setlocal` and `&l:` instead of `:set` and `&` unless you have explicit reason to do otherwise.

Style

When in doubt, treat vimscript style like python style.

Whitespace

Similar to python.

- Use two spaces for indents
- DO not use tabs
- Use spaces around operators. This does not apply to arguments to commands.

  ```
  let s:variable = "concatenated " . "strings"
  command -range=% MyCommand
  ```

- Do not introduce trailing whitespace. You need not go out of your way to remove it. Trailing whitespace is allowed in mappings which prep commands for user input such as "`noremap <leader>gf :grep -f`".
- Restrict lines to 80 colmns wide
- Indent continued lines by four spaces
- Do not align argumentes of commands

```
command -bang MyCommand call myplugin#foo()
command MyCommand2 call myplugin#bar()
```

Naming

In general, use `plugin-names-like-this`, `FunctionNamesLiekThis`, `CommandNamesLiekThis`, `augroup_names_like_this`, `variable_names_like_this`.

Always prefer variables with their scope plugin-names-like-this

Keep them short and sweet.

FunctionNamesLikeThis

Prefix script-local functions with `s:`.

Autoloaded functions may not have a scope prefix.

Do not create global functions. Use autoloaded functions instead.

CommandNamesLikeThis

Prefer succinct command names over common command prefixes.

variable_names_like_this

Augroup names count as variables for naming purposes.

Prefix all variables with their scope

- Global variables with `g:`
- Script-local variables with `s:`
- Function arguments with `a:`
- Function-local variables with `l:`
- Vim-predefined variables with `v:`
- Buffer-local variables with `b:`

`g:`, `s:`, and `a:` must always be used.

`b:` changes the variable semantics; use it when you want buffer-local semantics.

`l:` and `v:` should be used for consitency, future proofing, and to avoid subtle bugs. They are not strictly required. Add them in new code but don't go out of your way to add them elsewhere.