Style Guide

Not just about code, a style guide is a list of decisions we've made, that we want to be consistent about going forwards. It does not need to be comprehensive of all possible issues, nor does it need to confine itself to trivial topics such as formatting. It's a tool for ourselves so we don't forget where we've been, and can avoid solving the same problems again.

Don't be shy about adding to it. Things written here do impose some burden, but the hope is that they lessen other burdens in the long run. Use your judgement about what's worth it.

Please fix style issues in existing code as you encounter them. Style is aspirational, a journey not a destination. :)

- Language and terminology
- Code
- Operations

Language and terminology

Language

Most existing Lix documentation is written in British English. We intend to continue with that.

Terminology

(FIXME: unsure if this should be in the style guide but ... it kinda should be -jade)

- **Nix language** - Use this to refer to the language which haunts us all.
- **Nix** - Nix refers to the *technology*. Used when referring to the Nix store, for example. Or, to a Nix derivation. Lix is a *Nix* implementation.
- **CppNix** - This is the preferred term for when it is necessary to refer to Nix, the software that Lix is forked from, rather than the technology.
- **Lix** - Use *Lix* when referring to the implementation. For example, "Install Lix".
- `nix`, `nix-build`, etc - Use lowercase `nix` when referring to the `nix` command, which is still supported by Lix.

Code

Code changes

Tests

If at all practicable, all new code should be tested to some extent. If writing a test is hard, we need to prioritize making it easier, and potentially block features if that is the case.

Documentation

Reference documentation should be added, in addition to release notes (`doc/manual/rl-next-dev`), for user visible changes.

For notable dev facing changes, consider adding release notes in `doc/manual/rl-next-dev`. This is not critical for all changes; in some cases it may make more sense to write it up in dev documentation instead, and indeed it may be ok to defer writing that dev documentation (it's helpful to create an issue to not forget).

Benchmarking

Changes that touch the core of the evaluator or other performance critical code in Lix should be benchmarked.

See bench/README.md for instructions.

Changelist size

If a CL is too long to review, it should be split up into smaller pieces with tests. The exact length varies but passing the 1000 line mark should give significant thought to splitting.

- When a CL is split, each commit should still be a valid state (tests passing, etc). If you must, you can gate in-progress changes with a flag or similar until the final commit. (Qyriad)

Commit messages

Include at least a sentence or two as to why you are making a commit. For example, it can be nice to have the reproduction of a bug in the commit message. The commit message *is* the message for

your review.

There's no particular format or specific style for commit messages; just make sure they're descriptive and informative.

C++

While we hope to migrate the Lix interpreter from C++ to Rust eventually, C++ is a language that is likely to exist for a long time, and we may end up having to use it in other contexts.

Lix is a C++20 codebase. Features of C++20 that compile on all supported platforms can be used.

NULL vs nullptr

`nullptr` where at all possible.

Static vs anonymous namespace

Prefer anonymous namespace, both currently exist in the codebase (jade: any other opinions?).

Type Aliases with `typedef` vs `using`

Prefer `using` declarations, as they can be used in more places, can be templatized, and have clearer syntax. Both currently exist in the codebase. (Qyriad)

TODO/FIXME/XXX Comments

jade: this is not consistent with the conventions I use, needs further discussion imo (TODO: block in pre-commit hook, used in local tree but should never pass code review, FIXME(name||feature): its busted, someone should go fix it later, XXX: this is bad, we are writing down that it is ugly but leaving it as-is as we didn't figure out a better way)

Something along the lines of:

- **TODO**: acknowledgement that something is acceptably-for-now incomplete, especially if the scope of fixing it is high or unknown
- **FIXME**: this should be fixed before the feature or major change that it's a part of is considered "ready"
- **XXX**: this should not pass code review and should be considered a left-in mistake

Header files

Filenames

Headers should end with `.hh`. This reduces the likelihood anyone will try to include them from C files, which would require following the rules of both languages and is easy to get wrong.

The implementation of the functions declared in a `.hh` file should be in a `.cc` file of the same name, absent reasons to do otherwise.

Order-independence

Headers should not care what order they're loaded in.

The exception, for now, is `config.h` in the `lix` repo. This must always come before all other headers. This observation should not be taken to imply it must always be that way, but at the moment it's helpful to be aware of.

Idempotence

Use `#pragma once`, it helps. You can see this in most existing header files.

`///@file` and header documentation

`///@file` should be at the top of all nix headers - Doxygen and other tools use it to decide whether a header should have documentation generated for definitions in it. See the relevant Doxygen documentation for more details.

Strongly consider adding a description of the purpose of a header file at the top of it in with `@brief A sentence saying what it is for`.

Examples:

```
/**
 * @file
 * @brief This header is for meow meow cat noises.
 */
```

```
/// @file
/// @brief meow meow meow
```

Source files

Filenames

Source files should end with `.cc`.

Python

Python is a widely used tooling language in the Lix codebase. It's used in the test suite in `functional2`, a pytest based test suite intending to replace the Bash `functional` test suite. It's what the release engineering in `releng/` is written in.

In general, follow PEP-8 while writing Python code: identifier style should be the typical Python one per PEP-8, indents are 4 spaces, etc. In the near future, we will autoformat Python code with `ruff`.

Type annotations

Type annotations should be used where they are helpful, and should be used for parameter and return types of functions. If it is too much of a bother to convince the type checker of something, it is acceptable to put an Any there.

Prefer inserting Any if the alternative is lying to the type checker via `# type: ignore` comments.

`dataclasses` is a useful library that produces nicer, more typed, code, and it is widely used. It's the correct choice for most classes which are just blobs of data.

Doc strings

Please write them. Single-line ones are better than none at all.

We use the Sphinx docstring format for parameter documentation, though we currently do not generate Sphinx docs for our Python codebases. See this Sphinx tutorial for an example and the Sphinx domains reference for reference.

Nix language

Unsurprisingly Nix contains Nix code. Some amount is tests and a lot is packaging.

We use the nixfmt formatter on files outside the test suite. It's run through `treefmt` with `pre-commit` hooks. Nix code outside the test suite is expected to be formatted.

Test suite files need not be formatted with the formatter at this time, but please consider doing so with new tests that don't rely on formatting.

with

Prefer not to use `with` to bring things into scope as it obscures the source of variables and degrades language server diagnostics.

Use `let inherit (attrset) attrs` instead.

Meson

Generally based on the style in Meson's docs made consistent and with a couple tweaks; notably multiline function calls are done in "block style" (think like rustfmt does it), rather than aligned, e.g.:

```
executable('sdlprog', 'sdlprog.c',
  win_subsystem : 'windows',
  dependencies : sdl2_dep,
)
```

rather than:

```
executable('sdlprog', 'sdlprog.c',
           win_subsystem : 'windows',
           dependencies : sdl2_dep)
```

Meson's docs go back and forth on this, but we also put a space before and after the colon for keyword arguments (so `win_subsystem : 'windows'`, rather than `win_subsystem: 'windows'`).

Operations

Operational Conventions

Code Review

Self Stamping and Merging

On our Gerrit, core members have permissions to +2 any arbitrary CL, because sometimes we should be able to get something in quickly, and talk about it afterwards. In almost every case, the author of a change should not +2 their own CL, however Lix members may use their best judgement so long as they talk about it with the team when they can. Some cases where skipping synchronous review is a good idea:

- Reverting commits that accidentally broke main
- Fixing typos in other peoples' CLs that you would have +2'd, then +2'ing the edited CL
- Maybe typo fixes in `main`, though those can probably wait to be reviewed

Just make sure to talk about what you do :)