# Contributing to Leela Zero

## C++ Usage

Leela Zero is written in C++14, and generally encourages writing in modern C++ style.

This means that:

\* The code overwhelmingly uses Almost Always Auto style, and so should you.

\* Prefer range based for and non-member (c)begin/(c)end.

\* You can rely on boost 1.58.0 or later being present.

\* Manipulation of raw pointers is to be avoided as much as possible.

\* Prefer constexpr over defines or constants.

\* Prefer "using" over typedefs.

\* Prefer uniform initialization.

\* Prefer default initializers for member variables.

\* Prefer emplace\_back and making use of move assignment.

\* Aim for const-correctness. Prefer passing non-trivial parameters by const reference.

\* Use header include guards, not #pragma once (pragma once is non-standard, has issues with detecting identical files, and is slower https://gcc.gnu.org/bugzilla/show\_bug.cgi?id=58770)

\* config.h is always the first file included.

\* Feel free to use templates, but remember that debugging obscure template metaprogramming bugs is not something people enjoy doing in their spare time.

\* Using exceptions is allowed.

## Code Style

\* Look at the surrounding code and the rest of the project!

\* Indentation is 4 spaces. No tabs.

\* public/private/protected access modifiers are de-indented

\* Maximum line length is 80 characters. There are rare exceptions in the code, usually involving user-visible text strings.

\* Ifs are always braced, with very rare exceptions when everything fits on one line and doing it properly makes the code less readable.

\* The code generally avoids any pointer passing and allows non-const references for parameters. Still, for new code it should be preferred to a) put input parameters first b) use return values over output parameters.

\* Function arguments that wrap are aligned.

\* Member variables in a class have an m\_ prefix and are private. Members of POD structs don't and aren't.

\* Constants and enum values are ALL\_CAPS.

\* Variables are lowercase.

\* Function names are underscore\_case.

\* Classes are CamelCase.

\* Comments are preferably full sentences with proper capitalization and a period.

\* Split the includes list into config.h, standard headers and our headers.

If something is not addressed here or there is no similar code, the Google C++ Style Guide is always a good reference.

We might move to enforce clang-format at some point.

## Adding dependencies

C++ does not quite have the package systems JavaScript and Rust have, so some restraint should be excercised when adding dependencies. Dependencies typically complicate the build for new contributors, especially on Windows, and reliance on specific, new versions can be a nuisance on Unix based systems.

The restraints on modern header-only libraries are significantly less because they avoid most of the above problems.

If a library is not mature and well-supported on Windows, Linux \*and\* macOS, you do not want it.

This is not an excuse to re-invent the wheel.

## Upgrading dependencies

The code and dependencies should target the latest stable versions of Visual Studio/MSVC, and the latest stable/LTS releases of common Linux distros, with some additional delay as not everyone will be able to upgrade to a new stable/LTS right away.

For example, upgrading to C++17 or boost 1.62.0 (oldest version in a Debian stable or Ubuntu LTS release) can be considered if there's a compelling use case and/or we can confirm it is supported on all platforms we reasonably target.

## Merging contributions

Contributions come in the form of pull requests against the "next" branch.

They are rebased or squashed on top of the next branch, so the history will stay linear, i.e. no merge commits.

Commit messages follow Linux kernel style: a summary phrase that is no more than 70-75 characters (but preferably <50) and describes both what the patch changes, as well as why the patch might be necessary.

If the patch is to a specific subsystem (AutoGTP, Validation, ...) then prefix the summary by that subsystem (e.g. AutoGTP: ...).

This is followed by a blank line, and a description that is wrapped at 72 characters. Good patch descriptions can be large time savers when someone has to bugfix the code afterwards.

The end of the commit message should mention which (github) issue the patch fixes, if any, and the pull request it belongs to.

Patches need to be reviewed before merging. Try to find the person who worked on the code last, or who has done work in nearby code (git blame is your friend, and this is why we write proper commit messages...). With some luck that is someone with write access to the repository. If not, you'll have to ping someone who does.

Experience says that the majority of the pull requests won't live up to this ideal, which means that maintainers will have to squash patch series and clean up the commit message to be coherent before merging.

If you are a person with write access to the repo, and are about to merge a commit, ask yourself the following question: am I confident enough that I understand this code, so that I can and am willing to go in and fix it if it turns out to be necessary? If the answer to this question is no, then do not merge the code. Not merging a contribution (quickly) is annoying for the individual contributor. Merging a bad contribution is annoying for everyone who wants to contribute now and in the future.

If a contributor can't be bothered to fix up the trailing whitespace in their patch, odds are they aren't going to be willing to fix the threading bug it introduces either.

## "Improvements" and Automagic

Improvements to the engine that can affect strength should include supporting data. This means no-regression tests for functional changes, and a proof of strength improvement for things which are supposed to increase strength.

The tools in the validation directory are well-fit for this purpose, as

is the python tool "ringmaster".

The number of configurable options should be limited where possible. If it is not possible for the author to make rules of thumb for suitable values for those options, then the majority of users have no hope of getting them right, and may mistakenly make the engine weaker. If you must introduce new ones, consider limiting their exposure to developers only via USE\_TUNER and set a good default for them.

## GTP Extensions

GTP makes it possible to connect arbitrary engines to arbitrary interfaces.

Unfortunately GTP 2 isn't extensive enough to realistically fit all needs of analysis GUIs, which means we have had to extend it. The lack of standardization here means that Go software is continously catching up to the chess world, especially after UCI was introduced. We should aim to make this situation better, not worse.

This means that extensions have the possibility of outliving Leela Zero (or any GUIs) provided they are well thought out.

It makes sense to be thoughtful here, consider the responsibilities of both GUI and engine, and try to come up with flexible building blocks rather than a plethora of commands for very specific use cases.

Experience and previous discussions can help understanding:

\* lz-analyze "avoid" and "allow" were added in pull request [#1949](https://github.com/leela-zero/leela-zero/pull/1949).

\* lz-analyze got a side-to-move option in pull request [#1872](https://github.com/leela-zero/leela-zero/pull/1872) and [#1642](https://github.com/leela-zero/leela-zero/pull/1642).

\* lz-analyze got a "prior" tag in pull request [#1836](https://github.com/leela-zero/leela-zero/pull/1836).

\* lz-analyze was added in pull request [#1388](https://github.com/leela-zero/leela-zero/pull/1388).

\* lz-setoption was added in pull request [#1741](https://github.com/leela-zero/leela-zero/pull/1741).

\* Pull request [#2170](https://github.com/leela-zero/leela-zero/pull/2170) has some discussion regarding how to navigate SGF

files that were parsed by the engine via GTP.