Document number P0559R0
Date 2017-01-31
Project ISO JTC1/SC22/WG21: Programming Language C++
Authors JC van Winkel <jc@oreo.nl>,
Jose Daniel Garcia <josedaniel.garcia@uc3m.es>,
Ville Voutilainen <ville.voutilainen@gmail.com>,
Roger Orr <rogero@howzatt.demon.co.uk>,
Michael Wong <fraggamuffin@gmail.com>,
Sylvain Bonnal <sbonnalc@gmail.com>
Audience: WG21

Operating principles for evolving C++

Preamble

Motto: Living languages must change, must adapt, must grow.
(quote from Edward Finegan, used in the 2005 addendum of the Japanese D&E)

It is important for the C++ community, in particular large companies that have invested heavily in C++, that C++ stays alive. It is very hard to hire good designers and programmers and hiring is almost impossible if the language of choice is perceived to be at the end of its development ("dead"). There are worries in some national bodies that C++ is not making enough significant progress, giving the risk of becoming a language that seems less "alive", heading towards an accelerated death by paralysis, lack of innovations that tackle modern challenges, and by thousands cuts of small isolated features.

We also observe that there is an inflow of new members of the committee. That is good. We lack a guideline document that lays down general design principles; some of which appeared in Bjarne Stroustrup's emails and other writings (including D&E) over the years, but we don't have an up-to-date document that enumerates design and evolution principles for the committee post-C++11. This document attempts to provide those principles in a form that is kept up to date and is easy to refer to and consult.

We suggest a set of principles to make explicit the mindset that so far only "floated in the ether" in order to keep C++ alive, healthy and progressing. It sets "rules" that will allow a more rapid evolution of the language while keeping language design coherent, manageable, and consistent with the fundamental principles of C++. Finally, the rules will make it easier to set priorities and give resolution to conflicting potential new features.

We should keep in mind:

- We will make errors
    - Make them early so that we can fix them
- We aim to maximize successes
    - Rather than minimizing failures
- Integrate early
    - And be willing to back out if wrong
- Be confident
    - On average we have succeeded
- Delaying a decision is a decision
    - Delays often imply increased complexity and decreased coherence
- Any change carries risk
    - Doing nothing is also risky
- Don't confuse familiarity and simplicity
    - Such confusion hinders and delays major improvements

Principle 1. Introduction

This document contains the recommended operating principles of the SC22/WG21 C++ standards committee.

We expect not to violate these recommended principles. The desire is that these principles are agreed on by the majority of the committee, and that we take them into consideration when changing or extending the language and its standard library.

Principle 2. Language design ground rules.

Changes made to the working paper of the standard should be in line with the original C++ language design rules as listed in Appendix A.

Principle 3. Procedures (non-ISO dictated)

WG21 has been split up into 4 working groups: Library Evolution (LEWG), Library (LWG), Evolution (EWG) and Core (CWG).

- We have a train release model where features are added to the working paper when they are ready.

- Technical Specifications (TS) need less rigorous quality/readiness before going out for Balloting than the International Standard.
- When putting out a TS, a list of questions should be prepared that need to be answered before merging the TS into the C++ working paper. When the questions have been answered, the effort to merge the TS into the C++ WP should get high priority
  - Rationale: this prevents TSes from lingering on forever and never making it into the C++ working paper. By default, the committee should aim for 'high priority' features to land in the Working Draft.
- When considering whether a feature should be a normal proposal or a separate TS: TSes do give extra overhead, such as extra balloting. An indication can be if you say something like "Feature X is special because…". Example: Transactional Memory is special because of hardware support still being in flux.
- If an almost ready "grand" feature doesn't make it for an IS, integrate it in the next Working Paper as soon as possible so
  - the committee can build on it in the next version
    - e.g. concepts will allow support for the library to use concepts.
  - do not risk bikeshedding near the end of the cycle and thus missing yet another train
- Two conflicting TSes may be active at the same time. That is not a problem - TSes are there to experiment and gain experience.
- Grand project ideas have priority over smaller ones when it comes to conflicts.
  - To prevent small items being a road block for large features
- If a working group (LWG or CWG) had a strong consensus, the motion is put forward and WG21 should generally trust the subgroup.
  - If people who were not present during the working group discussion have concerns, they should ask the subgroup participants before plenary or trust the subgroup in plenary; plenary is not a forum for extended technical discussion, and there is no provision for "hold up the feature until I have time to weigh in." In plenary, questions of the form "did you consider XYZ" are fine and can be answered with Yes or No and a summary by someone knowledgeable but without real-time technical discussion.
  - People who were present during the subgroup discussion, but were in the small minority, had their concerns fully heard in the subgroup. In plenary they would speak up to persist in raising their concerns only if the issue is so serious that it would want them to vote No on the whole standard if this decision was adopted, and their concern is shared by others in their national body so that it is a national body concern and not an individual concern.
- If a subgroup had only a weak consensus, the subgroup chair reports that so the committee can take that into consideration, and optionally a representative of the majority and of the dissenting minority may each present a well-prepared concise technical summary of the concerns without technical discussion from the floor, taking only clarifying questions so the room understands the presented issues but does not debate it.

- "Over-my-dead-body" votes on WG21 level need to be followed by a paper describing the concerns in the post-meeting mailing so they can be talked about and weighed just like new proposals are.

4. Culture

- We are a diverse group of people with many backgrounds socially and culturally. Be aware of that.
- Social diversity matters: https://www.scientificamerican.com/article/how-diversity-makes-us-smarter/
- Be respectful of one another. Our basic starting point is that we are all seeking to make C++ better, even when we do not agree.
    - Always use respectful language
    - Do not speak over other
    - Listen more than talk
    - Keep to only what you know and reduce speculation

5. Acknowledgements

Thanks to Gabriel Dos Reis for assistance and constructive comments on this document.

Appendix A. Language design ground rules.

Bjarne Stroustrup in his Design and Evolution of C++ has written down the ground rules he used for developing C++. These are the rules he summarized in the foreword of the 2005 Japanese D&E:

1. express ideas directly in code
2. express relations among ideas directly in code
3. express independent ideas in independent code
4. compose code representing ideas freely wherever the composition makes sense

In his later document http://www.stroustrup.com/crc.pdf, (summarizing D&E) he makes this more concrete::

- C++ Design Aims
    - C++ makes programming more enjoyable for serious programmers.
    - C++ is a general-purpose programming language that
        - is a better C
        - supports data abstraction

        - supports object-oriented programming
        - supports generic programming
- Design Principles
    - General rules
        - C++'s evolution must be driven by real problems.
        - C++ is a language, not a complete system. Don't get involved in a sterile quest for perfection.
        - C++ must be useful now.
        - Every feature must have a reasonably obvious implementation.
        - Always provide a transition path.
        - Provide comprehensive support for each supported style.
        - Don't try to force people.
    - Design-support rules:
        - Support sound design notions.
        - Provide facilities for program organization.
        - Say what you mean.
        - All features must be affordable.
        - It is more important to allow a useful feature than to prevent every misuse.
            - enable good programming rather than to eliminate bad programming
        - Support composition of software from separately developed part
    - Language-technical rules:
        - No implicit violations of the static type system.
        - Provide as good support for user-defined types as for built-in types.
        - Locality is good.
        - Avoid order dependencies.
        - If in doubt, pick the variant of a feature that is easiest to teach.
        - Syntax matters (often in perverse ways).
        - Preprocessor usage should be eliminated
    - Low-level programming support rules:
        - No gratuitous incompatibilities with C.
            - As close as possible, but not closer
            - Avoid gratuitous surprises in the name of C compatibility
        - Leave no room for a lower-level language below C++ (except assembler).
        - What you don't use, you don't pay for (zero-overhead rule).
        - If in doubt, provide means for manual control.

Added to that, we also have the following guidelines (most from Bjarne Stroustrup's presentation in Berlin, November 2016):

- Direct access to hardware
- Zero-overhead abstraction
- Stability and portability

- Make simple things simple
- Avoid 'compiler magic' when possible
- Prefer library solutions over language changes if feasible
- No language is perfect
  - For everything
  - For everybody
- Prefer generality over specificity: prefer standardizing general building blocks on top of which domain-specific semantics can be layered, as opposed to domain-specific facilities on top of which other domain-specific semantics can't be layered
- Concise expression of ideas