Doc. No. P3573R0
Date: 2025-01-12
Audience: SG21, EWG, LEWG
Authors: Michael Hava
J. Daniel Garcia Sanchez
Ran Regev
Gabriel Dos Reis
John Spicer
Bjarne Stroustrup
J.C. van Winkel
David Vandevoorde
Ville Voutilainen
Reply to: bjarne@stroustrup.com

Contract concerns

A suitable contract mechanism would be a boon for C++, but only one that could be reliably used by a very wide range of C++ users. The authors of this note have grave concerns about the current design (P2900, the so called MVP == Minimal Viable Product) and its direction.

The purported purpose of the MVP is to provide a base that provides a core set of functionalities that is known (to the extent possible) to be correct, while not forcing upon users facilities not known to be correct, or not appropriate for their use.

This is a list of concerns, one paragraph per concern, rather than an extensive discussion of each. The order is random:

- **Constification**: if you want to prevent modifications, “constification” does a poor job. It catches a few bugs from a major language change. Any special case can catch a few errors, but every special case comes with a large cost to implementors, users, and teachers. It is yet another irregularity to be surprised by. A warning can catch most problems coped with by constification.
- **Exception capture**: No, throwing an exception from a contract predicate is not a contract failure – the contract was never entered. A predicate that throws simply throws and should be handled as all other throws, not by a special case in contracts. The need to enclose predicate evaluation in try-blocks may or may not impose costs on various implementations.
- **Overly complex hierarchy contract model**: It is untried. Don’t try to force everyone into a single untried design. Design is to make decisions, and if we are not assured that our design is correct, we should defer such a decision. The model with four or more contracts evaluated upon entry and exit from a virtual function is a solution to a subset

of the problems. This solution is known to be incomplete and is also not known to be correct. Also, there are many important designs that require contracts to be inherited.

- **Instability of overall design**: There were 10+ papers suggesting changes, 1,000+ reflector messages, and dozens of changes to P2900 in 2024. It is a full-time job to keep up. Nobody outside a small group of people knows what is really being proposed. This is not a solid basis for an international standard.
- **Far too much is implementation defined**: The average to expert user will have a hard time to figure out what's checked and what's the result of a failure. Writing portable contracts is hard with key issues are implementation defined. It is essentially impossible to explain/teach to a beginner what a contract does. At least **assert()** can be explained to a novice in about one minute.
- **No grouping of contracts**: Enabling/disabling contracts is all or nothing. There is no way to sorting contracts into groups that can be separately enabled and disabled.
- **Defaults**: P2900 claims to have a sweet spot for its "default" syntax, but it has not been demonstrated that we have the right defaults (e.g., we might want to save that for a future "safe" mode, exceptions in predicates, virtual functions, etc.).
- **Untried**: there is implementation experience (which is good), but
    - Preconditions and postconditions have not been tried at scale. More specifically, preconditions and postconditions using the proposed virtual function mechanism are almost completely untried.
    - There is no usage experience with mapping contract violations into exceptions.
    - the ability to link code with varying checking modes is poorly specified and untried.
- **Run-time vs. static checking:** The relationship between run-time checks and the use of contract predicates in static analysis seems weakly explored. For example, how do we specify predicates that cannot be executed?
- **Pointers to functions:** Seems a major feature, especially in C-style programming, to leave out.
- **Safety:** We don't yet know how contracts interact with safety (e.g., with profiles as promoted by SG23 offering opt-in to general guarantees). "Safety" must involve guarantees over a code base or parts thereof. Contracts primarily offer checking of correctness where it is used. Both require violation handlers, and it would be ideal if their violation handling was the same. We should not lock in important design criteria that affect safety without having resolved these issues (which related to "defaults" above).
- **Undefined behavior**: Should UB be directly addressed in the contract design or left as a problem for other proposals to deal with in general?

- **Composition of TUs**: It seems that the effect of linking together TUs with different contract settings is not well specified. In particular, if a template is instantiated in two Tus with different contract settings, do they get different settings? Is the linker supposed to prevent that? And if not, what determines which settings they get? Same questions for inline functions, **constexpr** functions, **consteval** functions, and concepts.
- **Contracts and Modules:** The same question as for TUs. Has the contract design even been tested with modules?
- **Contracts and static-reflection**: The same question as for TUs.
- **Compatibily of future improvements:** It would be nice if we could say that these issues can be handled in the future within the framework of the current design (a fundamental aim of the MVP idea), but it seems that the resolution to some of the issues would require incompatible changes.
- **The proposal is far too large:** Contracts should be a simple feature (as it is in some other languages), yet the specification is huge, the implementation is non-trivial, and there appears to be a steep learning curve for some features.
- **Scalability:** the proposal may not scale as it requires compilers and linkers to carry much information into the binaries. This implies changes to several tool chains.

These issues must be addressed, their resolution must be understood by a much larger group than the permanent members of SG21, and stable for a while after before becoming standard. We seriously doubt that can be done in time for C++26. As a proposal for an international standard, it is incomplete and untried.

It has been suggested that P2900 should be turned into a TS or a white-paper. If so, it should follow the DG's recommendation for TSs: do so only if such a paper is required to yield answers to a specific set of questions.

It would be ideal if the authors of this note could outline an alternative design. However, given the timescale and the long history of contract proposals (of which several of the authors were deeply involved) and modifications to P2900, we don't see a simple design that is likely to reach consensus. Remember what happened to the C++20 contract design.