The Need for Design Policies in WG21

Document #: P2979R0
Date: 2023-10-13
Project: Programming Language C++
Audience: Evolution Working Group
Library Evolution Working Group
Reply-to: Alisdair Meredith
<ameredith1@bloomberg.net>
Harold Bott
<hbott1@bloomberg.net>
John Lakos
<jlakos@bloomberg.net>

Contents

1 Abstract

Two risks inherent in large-scale collaborative projects are inconsistent design decisions and the time lost revisiting previous decisions when developing new features. The work of WG21 is a prime example of such a software-design project, including a Standard Library intended for wide reuse. These risks motivate the proposed creation of a standard framework and process for creating a readily accessible curated suite of reusable design decisions, *design policies*, that can be used to normalize and streamline the process of C++ Standards development.

In this paper, we will identify some past attempts to record policy decisions, noting instances of both successful and unsuccessful outcomes. We then sketch a methodology to develop a regular pattern by which to render (format), readily access, and thus effectively *memorialize* design policies, such as the criteria under which certain language features (e.g., `constexper`, `explicit`, and `noexcept`) are to be used in C++ Standard Library functions. This uniform rendering of design policies will turn out to be closely associated with a particular solution-evaluation process, *principled design*, which is predicated on identifying governing principles, prioritizing them, and then using this ordered sequence of principles to evaluate – on a per-principle basis — the various candidate solutions. Finally this paper provides a roadmap for two future papers that will more fully delineate what we mean by *principled design* and *principled-design policies*.

2 Revision History

R0 October 2023 (pre-Kona mailing)

Initial draft of this paper.

3 Introduction

Everyone participating in WG21 does so with the goal of producing the best possible Standard for C++. That said, not everyone will arrive at their design decisions based on the same principles. Unless all relevant principles are first agreed upon, design discussions can easily reach an impasse. In cases of agreement, we can later discover that a majority vote — even one with strong consensus — was uninformed regarding one or more important and relevant design principles.

Consistent application of *principled design* — a principles-first design-choice-resolution methodology — has been demonstrated to effectively help resolve difficult design problems. We propose that WG21, rather than voting on one of a set of solutions immediately after discussing the related problem, follow our *principled-design* protocol and thus first articulate and prioritize the relevant design principles.

Once such priorities are set, nonviable solutions can often be eliminated quickly, and the remaining candidate solutions evaluated against the ordered set of priorities to establish consensus for a winner. Using this system, we will avoid the inconsistency and inefficiency that occur when two or more solutions are arguably equally good, leading to repetitive discussion and voting, with results dependent upon which members are present. We assert and will demonstrate that a principled-design decision-making process has led, and will continue to lead, to far greater consistency than our typical *ad hoc* methods.

Moreover, once a fully informed principled-design decision is made, documenting that resolution *as it is made* would streamline any future discussion of that or similar design issues by avoiding reconsideration unless new circumstances or information warrants it.

This paper outlines high-level parts of what we propose to be a standard process for establishing reusable *design policies*, initially targeted for LEWG. In particular, we set the stage in this paper for two future proposals relating to (1) a principled approach to arbitrating contentious design decisions [P3004R0] and (2) a systematic mechanism for documenting and accessing established design policies [P3005R0]. Via these proposed techniques, we can get decisions right the first time and make them readily accessible, thus enhancing WG21's efficiency and consistency for its C++ standardization process for C++26 and beyond.

4 Previous Work

WG21 has attempted to enact and document design policies in the past. An incomplete list of preceding work would include the following.

— [P1000R5], "C++ IS schedule" — Herb Sutter provides the policy that defines our current "train" model of delivering a new Standard every three years.

— [P0592R5], "To boldly suggest an overall plan for C++26" — Ville Voutilainen provides a plenary-approved priority plan for each new revision of the Standard.

— [P2000R4], "Direction For ISO C++" — The Direction Group provides principles to guide the work of WG21 as a whole.

— [P1369R0], "Guidelines for Formulating Library Semantics Specifications" — Walter Brown provides a principled update for how to consistently specify Standard Library wording as the Standard has evolved.

— "LWG Reviewing Guidelines" — The LWG presents the oral history of how LWG applies consistent patterns when reviewing wording.[1]

We will discuss such previous work in more detail in our second follow-up paper, [P30005R0].

5 Applying Design Policies to the Library Evolution Process

The intended reuse of a fundamental library, such as specified by the C++ Standard, implies a need for maximal stability. An incorrect or inconsistent decision can result in enormous and often irreparable negative consequences. Despite our efforts to make the best decisions, we often find that a close-call decision could depend merely on who happens to be voting at the time.

Sometimes, however, the decision should have been clear, but key people were absent, and the general guiding design principles and policies that had been discussed previously and applied to similar issues were never considered. This lack of easy access to prior discussion and decision-making has led to numerous inconsistencies that we had to address over the years as we designed the C++ Standards. As a classic example, adding an inconsistent `noexcept` specification to a defaulted function in C++11, 14, and 17 has different behaviors in each of these standards: In C++11, it was considered ill-formed; in C++14, it caused the function to be deleted; and in C++17, it was treated as an override. The principles that reigned were (1) that the semantics could be implemented and (2) that it did something useful in practice.[2]

An important and misunderstood example of a principled-design policy is that of the Lakos Rule, which strongly advocates that the `noexcept` specifier not be mandated in the Standard (1) for any function having a narrow contract, (2) unless a client has some compelling need, in a generic context, to query the exception specification of a function for the purpose of laying down a more efficient algorithm at compile time. This policy, which was adopted with strong consensus by the entire committee in Madrid (March 2011), remained in effect until a straw poll in LEWG in Prague (February 2020) narrowly overturned it. The opportunity to reinstate it eventually presented itself at Varna (June 2023), but, by that time, several components having narrow contracts had been inconsistently (and therefore wrongly) given a `noexcept` specifier.

The proposed governing principles included mandating a consistent specification, minimizing object size, maximizing runtime performance, requiring better documentation, enabling the creation of backward-compatible extensions, allowing implementers flexibility in the quality of the implementation to trade off (1) reducing code size versus (2) not blocking thrown exceptions, and so on.

Once the claim that `noexcept` itself is a runtime optimization was debunked and it was established that `noexcept` on narrow contracts did undesirably interfere with a throwing exception handler for contracts, the entire joint session of LWG and LEWG at Varna agreed, with seemingly overwhelming consensus (i.e., by verbal acclamation; no poll was taken), that the Lakos Rule is an essential design policy and, if anything, not restrictive enough.

[1] https://wiki.edg.com/bin/view/Wg21PersistentInformation/LWGReviewingTips

[2] *Embracing Modern C++ Safely*, section 3.1, "`noexcept` Specifier," pp. 1085-1152, specifically footnote 2, p. 1086.

Starting from zero and reconsidering a library design choice, such as whether a particular kind of function should be declared `explicit`, `constexpr`, or `noexcept`, without first understanding the relevant principles that led to the previous *policy* decision, is inefficient and results in inconsistencies, which — even in the rare case of otherwise equally good solutions — are inherently suboptimal.

We assert that a standard framework for documenting *design policies* — vetted design patterns established through consensus resulting from a principled-design process (see below) — is essential for correct, consistent, and efficient design and will play a crucial role in establishing clear priorities, providing guidance, and ensuring consistency in the decision-making and implementation processes.

6 Applying Principled Design to the Language-Evolution Process

Before we can capture policies in LEWG, we first need to formally delineate a standard process by which we, as a group, can reliably decide how to formulate the best policies. As part of the process of developing the minimum viable product (MVP) version of Contracts for C++26, we have been faced with many unusually complex design decisions, some of which have been open questions for more than a decade. To demonstrate the power of our approach to principled design, we present two such decisions that might still be openly questioned without principled design.[3]

1. If a `noexcept` function has a precondition, and evaluation of that precondition throws an exception, should that exception propagate to the caller of the function or invoke `std::terminate()`?[4]

   A primary motivation for the Lakos Rule was to allow a function having a *narrow contract* — i.e., one with preconditions — to detect an out-of-contract call and optionally throw an exception (one known valid use being negative testing[5]). Declaring a function having a narrow contract `noexcept`, however, would obviously bar an exception from escaping the evaluation of that function. This restriction would apply even if the exception came from the evaluation of an assertion of the preconditions on the function.

   The C++20 (and C++26 MVP) contract-checking facilities allowed both precondition and postcondition decorations on the function declaration itself. If we allow the precondition and postcondition checks to occur on the outside of the boundary of the `noexcept` specifier that applies to the function invocation, we could potentially throw back to the client when a contract-checking annotation (CCA) identifies that a contract has been violated. Is this a good idea, and why?

   Allowing exceptions to propagate might seem like a naturally useful step to take; client code could recover from contract violations without needing to worry about unexpected program termination, and unit testing narrow contracts would become straightforward to accomplish. The ramifications of this decision, however, are hard to evaluate without enumerating the basic principles that might apply and seeing whether the decision is consistent or in violation of those principles.

   As described in detail in [P2834R0] and later revisited in [P2932R1], Contracts in the C++ language must adhere to the following principle if they are to remain viable tools for detecting defects: Changing whether a CCA is *checked* or *unchecked*, i.e., the CCA semantic[6] with which the CCA is evaluated, must not alter the compile-time semantics of a program. If enabling or disabling CCAs results in changes to program semantics, then we open the door to having a production defect (in a build with all CCAs disabled) that magically goes away when we turn contract-checking *on*.

   Allowing exceptions to propagate from a CCA on a `noexcept` function — but only in a checked build — would necessarily mean that the result of the `noexcept` operator would change as well, leading to potentially vastly different code paths taken in surrounding code. A common example of this highly

[3]The following two examples are taken from [P2834R0] and elaborated on in [P2932R1].

[4]Note this question is distinct and independent from whether the evaluation of the preconditions and postconditions should be implemented on the client side, implementation side, or both.

[5]Pablo Halpern and Timur Doubler CppCon'23, "Noexcept? Enabling Testing of Contract Checks," https://cppcon2023.sched.com/event/1Qtgc/noexcept-enabling-testing-of-contract-checks

[6]The Contracts MVP proposes three potential semantics. The *ignore* semantic does nothing and is considered *unchecked*. The *observe* and *enforce* semantics evaluate the CCA's predicate and invoke a violation handler on violations, and both are considered *checked*. See [P2900R1] for details on the current MVP.

undesirable scenario would be whether a `vector` copies or moves its elements on some operations based on whether the move operation is `noexcept`.

Given that propagation of exceptions thrown as the result of a failed precondition (or postcondition) would significantly alter such statically observable properties, we then know that such a decision would violate an *imperative* design principle (viz. Contracts do not change the meaning of a valid program), one that is absolutely essential for Contracts to fulfill their goals. Thus, principled design guides our hand to choose to always consider CCA evaluation to be within the `noexcept` boundary of that function — including even those tempting `pre` and `post` CCAs, which are attached to a function declaration.

2. When a local variable is referenced from within a lambda, that variable must be captured, either by reference or value, within the lambda closure object. If the predicate of a CCA is the only place within a lambda where a variable is referenced and thus is the only source of a need to capture the variable, then we potentially have a few choices.

   a. Add the implicit capture to the closure object only when we build the program in a way that would evaluate and check the CCAs (sometimes called a *checked* build).
   b. Add the implicit capture to the closure object independently of whether the CCA's predicate will be evaluated, i.e., in all potential build configurations.
   c. Make such a use of a local variable ill-formed.

   Again we must ask what principles apply to this question.

   The same principle that clearly determined our choice for exception propagation through `noexcept` — that the semantics of a CCA must not alter the compile-time semantics of a program — immediately rules out option (a).

   Another important principle for Contracts, as described in [P2932R1], is what we call the zero-overhead principle. This principle again comes from a motivation that a Contracts facility that does not detect bugs is a failed Contracts facility. By extension, a Contracts facility that does not become ubiquitously used is a failed Contracts facility.

   To be used reliably with no need for second thoughts, a successful Contracts facility must, when turned *off*, have absolutely zero runtime overhead on a program. Any implicit capturing of variables due to a CCA predicate would introduce the overhead of initializing and destroying the member variable of the closure object as well as increasing the size of the lambda's closure object. Both of these overheads, though typically small, can subtly become very large in otherwise reasonable situations. Perhaps more importantly, that we cannot state emphatically that no runtime overhead is implied merely by installing Contracts would give rise to FUD, which could easily slow the rapid adoption of Contracts. Therefore, option (b) has prodigious drawbacks and hence should probably not be considered viable either.

   This leaves option (c) as our only remaining choice. Option (c) also fits well with another principle for Contracts described in [P2932R1]: Make undecided behaviors ill-formed.[7] This principle indicates that, when no consensus is reached on a path for a particular decision or when the optimal solution for a particular problem remains unclear, choose to make ill-formed those programs that would need a definition for the undecided behavior. This choice allows compilers to experiment with potential solutions as conforming extensions and future Standards to settle on a better choice if ill-formed turns out to be wrong.

   Therefore, option (c) wins on two counts: It is the only truly viable option that does not violate one of our strong fundamental principles for Contracts, and should it be the wrong choice, it naturally leaves open a fully backward-compatible evolutionary path to add a well-formed behavior to implicitly capture from CCA predicates in the future.

The purpose of this paper, then, is twofold: Standardize a process for (1) making systematic design decisions that lead to the overall best solution to a given problem based on what we all agree are the most important governing principles, and (2) capture, document, and make readily accessible these decisions along with how the

[7]This principle is a special case of the more general form, which states that given two seemingly equally good alternatives, select the one that, if you are wrong, is easiest to migrate to the other.

design process was applied, including the relevant principles (along with any established relative priorities) upon which the resulting policy is based.

7 Roadmap

Given the needs we describe in this paper, we anticipate two follow-up papers.

7.1 "*Principled Design* for WG21," [P3004R0]

This paper (expected 23Q4) will delve into the concept of *principled design* — in the general context of the WG21 — and its role in solving design problems, presenting a framework that identifies objective and verifiable *imperative* principles to guide solutions, while also admitting for consideration more subjective and controversial ones. We will describe how we collect relevant principles, categorize and order them, and then eliminate obviously nonviable solutions. After that, we will discuss how to evaluate the remaining candidate solutions (and their variants) against a priority-ordered sequence of principles, leading to highly effective, tabular rendering, for ease of comparison.

After thoughtful consideration, one solution will ultimately emerge as a clear best answer for moving forward. Note that the degree and detail to which this process is applied is adjustable to fit the needs of each particular application.

7.2 "Documenting *Principled-Design Policies* for WG21," [P3005R0]

This paper (expected 24Q1) will propose a framework for capturing and maintaining policies within WG21 but focusing (at least initially) on the use of language features in library components. Policies serve as approved criteria for employing specific features, such as attributes, allocators, `explicit`, `constexpr`, and `noexcept`. The framework aims to establish a consistent and generalized approach that can be expanded over time to include almost any recurring decisions that we might find ourselves making, especially when the decisions encompass business, economic, or otherwise nontechnical situations. Each policy will follow a regular form, consisting of a name, purpose, concise wording, goals, relevant principles, and their imputed priority order. This proposed framework emphasizes the importance of stability, consistency, safety, performance, clarity, ready accessibility, and maintainability in policy development.

8 Conclusion

Software engineering typical comprises many trade-offs that can sometimes be highly nuanced in nature. We have identified a pressing need to reimagine how design decisions are made throughout WG21 to motivate a more consistent, more accessible way to document *design policies*, such as the recommended idiomatic use of specific language features in Standard Library components. Closely associated with this uniform rendering, we propose a *principled-design* process: Determine relevant principles, prioritize those principles to establish a priority-based sequence, evaluate each solution against that ordered list, tabulate the results, select an optimal solution, and *memorialize* it as as an easily accessible principled-design policy. Finally, we offer a road map for producing two additional papers:

1. [P3004R0], "*Principled Design* for WG21"
2. [P3005R0], "Documenting *Principled-Design Policies* for WG21"

Principled design is a valuable tool that helps us to inform and focus our thinking but is not in itself a substitute for intelligence, experience, or good taste. By advocating for a *principles-first* design process for capturing and maintaining *design policies*, we will create a sound, consistent, easily accessible, maintainable, collectively curated repository of reusable software capital.

9 Acknowledgments

The authors would like to thanks Joshua Berne, Mungo Gill, and Mike Verschell for providing initial engineering and managerial feedback on this proposal. Thanks also goes to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

Lastly, we'd like to thank Lori Hughes for reviewing this paper and providing editorial feedback.

10 References

[P0592R5] Ville Voutilainen. 2022-10-15. To boldly suggest an overall plan for C++26.
https://wg21.link/p0592r5

[P1000R5] Herb Sutter. 2023-05-10. C++ IS schedule.
https://wg21.link/p1000r5

[P1369R0] Walter E. Brown. 2018-11-25. Guidelines for Formulating Library Semantics Specifications.
https://wg21.link/p1369r0

[P2000R4] Roger Orr, Howard Hinnant, Roger Orr, Bjarne Stroustrup, Daveed Vandevoorde, Michael Wong. 2022-10-15. Direction for ISO C++.
https://wg21.link/p2000r4

[P2834R0] Joshua Berne, John Lakos. 2023-05-15. Semantic Stability Across Contract-Checking Build Modes.
https://wg21.link/p2834r0

[P2900R1] Joshua Berne, Timur Doumler, Andrzej Krzemieński. 2023-10-15. Contracts for C++.
https://wg21.link/p2932r1

[P2932R1] Joshua Berne. 2023-10-15. A Principled Approach to Open Design Questions for Contracts.
https://wg21.link/p2932r1