Memorializing Principled-Design Policies for WG21

Document #:	D3005R0
Date:	2024-02-08
Project:	Programming Language C++
Audience:	EWG, LEWG

Reply-to: John Lakos
<jlakos@bloomberg.net>
Joshua Berne
<jberne4@bloomberg.net>
Harold Bott
<hbott1@bloomberg.net>
Mungo Gill
<mgill83@bloomberg.net>
Alisdair Meredith
<ameredith1@bloomberg.net>
Bill Chapman
<bchapman2@bloomberg.net>
Mike Giroux
<mgiroux@bloomberg.net>
Oleg Subbotin
<osubbotin@bloomberg.net>

Contents

1 Abstract

This paper proposes a process and framework for capturing and maintaining, with WG21, *policies*, i.e., sanctioned criteria for employing a specific language feature — such as `explicit`, `constexpr`, `noexcept`, attributes, and allocators — to the specification of constructs, such as functions and classes, defined within the C++ Standard Library.

The goals of any such framework must include mitigating discussion of recurring issues with each new Standard Library component considered. This framework emphasizes consistency, maintainability, and accessibility of the policies developed; minimizes the additional cost associated with developing and updating polices when needed; and ensures a provably optimal result.

Building on our principled-design methodology [P3004R0], the approach proposed here accompanies and expands upon Inbal Levi's policy development initiative in LEWG, and the authors hope to gradually expand the framework to facilitate all of WG21.

2 Revision history

PRE-PRINT — THIS DOCUMENT WILL BE FINALIZED FOR THE FEBRUARY 2024 MAILING

R0 February 2024 (pre-Tokyo mailing)

— Initial draft of this paper
— This paper is the third paper in the road map presented in [P2979R0]

3 Introduction

Committee time is a scarce resource and must be used efficiently. Each time we have a new *library* API (e.g., function, class, template), we'll need to decide how to accessorize it by pulling from an increasingly large suite of *language* constructs (e.g., `explicit`, `constexpr`, `noexcept`). But such recurring decisions are not limited to just new language features and extend to

— naming conventions
— how proposals should be formatted
— the kinds of libraries that should and shouldn't be part of the Standard

We need to make such decisions

— *consistently*, to maximize the C++ users' experience
— *efficiently*, to minimize unnecessary use of Committee time
— *correctly*, to maximize the effectiveness of C++ in practice

LEWG polices, as described here, would directly achieve all three essential goals at once.

We have long known that LEWG policies are sorely needed. In June 2023 in Varna, Bryce Adelstein Lelbach called and led an evening LEWG meeting to discuss LEWG policies theoretically. The motivation for a particular policy was brought up, and the meeting became contentious. Now Inbal Levi is LEWG chair and is pursuing the adoption of a standardized approach to policies for LEWG, which might someday expand to include more subgroups if not all of WG21.

Separately, we have also long known that we need a more structured approach for making technical decisions than a simple vote on a collection of solutions. In SG21, we've pioneered a *principled-design* approach, which arose when Joshua Berne observed that allowing a build mode to affect the semantics of a program is the antithesis of sound software design. This observation became an imperative principle — "Concepts do not see Contracts," i.e., local semantics do not change when contracts are added or are built differently — that has driven the design of the Contracts MVP from discordant discussion to a demonstrably optimal result for C++26.

3.1 Principled Design Concisely

Principled design is a methodical process for differentiating between candidate solutions to a design problem and identifying the optimal choice. [P3004R0] describes our principled-design methodology in detail; here we offer a concise summary.

A *principle* describes a favorable property or characteristic of a solution. Some principles are generally applicable, and others are specific to a given problem. Each candidate solution has associated principles, *motivating principles*, to recommend it. Each solution also has related concerns. The goal of principled design is to help informed readers identify, from a given set of solutions, those solutions that best satisfy our most important principles.

After discussion, the proposal author creates a clear and concise statement of the problem to be solved, its business justification, and measures of success. The author then describes each candidate solution in their anticipated order of preference.

— Each solution has a short title.
— The solution's description is concise but sufficient to distinguish it from other candidate solutions.
— Each solution includes a list of the motivating principles that favor that solution over others.
  — Each principle is stated as either a predicate or something to maximize or minimize.
  — The second and subsequent solutions do not repeat motivating principles listed in the first solution.
— The proposal author is honest and lists any concerns that might pertain to each solution, including the proposed solution.

Next is the process of collecting, curating, refining, characterizing, and ranking the principles. (Note that once the proposal author copies the motivating principles to begin the refinement process, they typically do not go back to update the original ones; see [P3004].)

— Collect just the motivating principles in the order in which they appear in the paper.
— Retain the order but modify the wording of the principles for precision, form (predicate or metric), and generality.
— Split overly coarse principles.
— Remove duplicates.
— Combine those that are sufficiently similar, replacing the first one and removing the rest.
— Add to the end any that were missed.

Let's suppose the proposal author has six refined principles.

Principle Statement
Maximize the intuitiveness of the syntax.
Maximize usefulness in practical applications.
Some way to implement the solution is known.
Maximize existing implementation experience.
Minimize changes to well-defined runtime behavior.
Do not preclude good uses to avoid misuse.

Note that phrasing a principle as a *metric* ("Minimizes" or "Maximizes") is preferable to phrasing it as a *predicate* (i.e., eliciting a yes or no answer). A reasonable compliance score for a metric is often be limited to the range [9 … 1], and either boundary condition (independently) may or may not be appropriate depending on the metric. For example, instead of phrasing the first principle as the predicate "The syntax is intuitive," we have instead chosen to phrase it as the metric "Maximize the intuitiveness of the syntax," which would reasonably be scored in the range [9 … 1] but not with @ or - since they would indicate an objectively provable boundary, which would be unreasonable for this metric. The principle "Maximize existing implementation experience" means that, as we score solutions having more implementation experience (e.g., more platforms, at larger scale, more users, longer calendar time in the field, etc.), they garners a higher compliance score in the range [9 … -] because some solutions might have literally zero implementation experience. Rather than "There is no change to well-defined runtime behavior," we prefer to say "Minimize change to well-defined runtime behavior," which would reasonably score in the range [@ … 1], with @ being an acceptable compliance score (meaning no defined behavior would change). Note that use of - here (meaning *all* defined behavior would change) would clearly be hyperbolic.

The next step is to assign each principle a terse identifier for later reference.

Principle ID	Principle Statement
intuitiveSyn	Maximize the intuitiveness of the syntax.
goodInPractice	Maximize usefulness in practical applications.
implementable	Some way to implement the solution is known.
implemented	Maximize existing implementation experience.
stableDefined	Minimize changes to well-defined runtime behavior.
maxOmniverse	Do not preclude good uses to avoid misuse.

The proposal author uses the following measurement scheme for assessing the *objectivity* and *importance* of each principle and, later in the *compliance table*, how well each solution satisfies each principle. Note that we use a single-character representation for compactness and visual acuity.

Symbol	Weight	Quality	Objectivity	Importance	Compliance
@	1.0	True	Provably	Imperative	Perfect
9	0.9	90%	n/a	High	High
8	0.8	80%	n/a	n/a	n/a
7	0.7	70%	n/a	n/a	Medium-High
6	0.6	60%	n/a	n/a	n/a

Symbol	Weight	Quality	Objectivity	Importance	Compliance
5	0.5	50%	Largely	Medium	Medium
4	0.4	40%	n/a	n/a	n/a
3	0.3	30%	n/a	n/a	Medium-Low
2	0.2	20%	n/a	n/a	n/a
1	0.1	10%	n/a	Low	Low
-	0.0	False	Subjective	Irrelevant	Not At All
?	n/a	Unknowable	n/a	n/a	Unknowable

The proposal author coarsely scores the principles for objectivity.

o	Principle ID	Principle Statement
-	intuitiveSyn	Maximize the intuitiveness of the syntax.
-	goodInPractice	Maximize usefulness in practical applications.
5	implementable	Some way to implement the solution is known.
5	implemented	Maximize existing implementation experience.
@	stableDefined	Minimize changes to well-defined runtime behavior.
5	maxOmniverse	Do not preclude good uses to avoid misuse.

Now the author repeats the process and scores importance.

i	o	Principle ID	Principle Statement
1	-	intuitiveSyn	Maximize the intuitiveness of the syntax.
9	-	goodInPractice	Maximize usefulness in practical applications.
@	5	implementable	Some way to implement the solution is known.
5	5	implemented	Maximize existing implementation experience.
@	@	stableDefined	Minimize changes to well-defined runtime behavior.
9	5	maxOmniverse	Do not preclude good uses to avoid misuse.

Once the principles are characterized, they are ranked, first by importance, second by objectivity, and third by a pair-wise comparison, which might occasionally override the other considerations, such as objectivity. Note that, in practice, we often equivalently sort the principles using the first two criteria, and then make linear passes swapping principles that we feel are out of order until we're satisfied.

Rank	i	o	Principle ID	Principle Statement
6	1	-	intuitiveSyn	Maximize the intuitiveness of the syntax.
4	9	-	goodInPractice	Maximize usefulness in practical applications.
2	@	5	implementable	Some way to implement the solution is known.
5	5	5	implemented	Maximize existing implementation experience.
1	@	@	stableDefined	Minimize changes to well-defined runtime behavior.
3	9	5	maxOmniverse	Do not preclude good uses to avoid misuse.

Notice that the characterization of importance and objectivity in the table above implied a total order for rank (with no need to break ties), and we were satisfied with the resulting order.

The next step is to map the solution titles to capital letters, i.e., A, B, C, for the column headers in our compliance table. For Standards proposals using principled design, A is, by default, the status-quo solution. (For *policy* proposals using principled design, A represents having no policy, and we'll discuss that more later.)

A. Status-Quo Default Solution
B. Proposed Solution
C. Alternate Solution
D. Alternate Solution
E. Alternate Solution
F. [continue listing alternate solutions]
G. Newly Proposed Solution
H. Another Newly Proposed Solution

Now the author creates the compliance table and then scores how well each solution satisfies each principle.

Rank	**i**	**o**	**Principle ID**	**A**	**B**	**C**	**D**	**E**	**...**	**G**	**H**
1	@	@	stableDefined	@	@	@	@	3		@	@
2	@	5	implementable	@	@	@	1	@		@	@
3	9	5	maxOmniverse	@	5	7	@	@		7	@
4	9	-	goodInPractice	-	9	5	9	9		9	9
5	5	5	implemented	@	@	7	-	5		7	7
6	1	-	intuitiveSyn	-	9	7	9	5		7	9

From the compliance table, the proposal author can conclude that solutions B and C have advantages and disadvantages with respect to each other, but neither is superior to the status-quo solution, A. Newly proposed solutions G and especially H consistently score better than A, B, or C on the most important principles. Barring any new concerns, the group would most likely select solution H as the optimal choice.

3.2 Inbal Levi's Initial Requirements for Policy Proposals

Let's now turn our attention to determining a policy to advise LEWG when applying a specific C++ language feature — such as `explicit`, `constexpr`, or `noexcept` — to functions defined in the C++ Standard Library.

Inbal Levi has requested that each policy statement submission include these important aspects.

— The policy itself should be concise and should suggest wording to the "List of Standard Library Policies" section in the SD-9 document.
— For each policy accepted, LEWG will add the concise rule to SD-9 and link to the paper from the SD-9 to provide the full information.
— The policy should have a detailed rationale (including examples and a description of the use cases affected).
— *When* the policy applies should be clearly described. (Examples will be beneficial as well).
— One of the goals is efficiency and friendliness to newcomers. An unclear policy will lead to confusion.

3.3 Employing Principled Design in a Policy-Centric Context

A policy has a succinct name, a clear statement, and a description that includes examples and exemptions. Members of LEWG also need to understand why this policy statement is better than some other, how to deal with exemptions, and how to propose a better policy. LEWG needs a work product that clearly shows how a particular policy statement was selected and how it satisfied the most important criteria relevant to the problem it addressed.

The goal of the principled-design process described in [P3004R0] is to methodically select the best option from several solutions to a problem. In the case of policies, however, the solution is not an answer to a single question, but an algorithm that can be applied to answer similar questions that will arise in many future Standards proposals.

Suppose the problem statement is "Develop a policy describing when to use `explicit` for constructors." Solution A will always be to have *no policy* because LEWG might revert to having no policy should new information reveal that to be the best choice. We will also note the current policy if one exists.

A proposal author will consider all the policy statements and propose the one they feel best about, calling it policy statement B, the proposed policy statement. All the remaining policy statements have at least one thing they do well and will be represented as C, D, E, and so on. Authors proposing policy statements will continue to follow this naming scheme.

LEWG, however, will use a more static naming scheme. Once LEWG has *adopted and published* a policy, the policy statement sequence *in LEWG-sponsored policy papers* will merely be extended in future revisions, i.e., F, G, H, and so on.[1] The LEWG-sponsored paper, with its ranked principles and compliance table, will act as a transcript of the decision and its rationale.

3.3.1 Proposing a New Policy for LEWG

If an author feels that a policy is needed where none currently exists, the process is straightforward. Each policy, to justify the Committee time to create it and the ongoing cost of having to adhere to it, is presumed to provide some substantial value over having no policy. The author must clearly demonstrate why having a policy statement would be helpful and why the policy statement is needed. Each policy will, therefore, have some sort of business justification, which will typically pull from several common motivations.

— Maximize consistency within the C++ language and its APIs.
— Maximize the effectiveness of the C++ language for practical applications.
— Improve efficiency of Committee time.
— Reduce risk of security vulnerabilities.
— Reduce risk of defective code.

The proposal itself will typically contain

— a concise statement of the problem (the design question the policy aims to solve)
— one or more well-described examples
— descriptions of any variations of the policy under consideration as needed

Once sufficient background material has been provided, the principled-design process can begin with a description of the proposed policy, including its title, precise policy statement, list of motivating principles (each with optional reasons), and a list of concerns. Each alternate policy is similarly described. If the policy statement proposed has competing policy statements in other papers, the proposal author would be wise to cite those papers and include those proposed policy statements as alternates, describing and scoring them as fairly and objectively as possible.

In the course of this research, several policy statements will usually arise, each with its own pros and cons. Some policy statements may be quite similar with only small variations. Some variations may apply across all policy statements, in which case Karnaugh maps[2] (which we'll use in our example) can often be employed to good advantage.

The proposal author then copies all the motivating principles in chronological order and curates, refines, characterizes, and ranks them. The author creates a compliance table and scores the policy statements and then analyzes the compliance table and makes recommendations. Note that this process does not produce a final answer but merely elucidates important objective features that lead to consistent and sound conclusions by informed human beings.

3.3.2 Processing Policy Proposals for LEWG

When a policy statement is proposed, the proposal author must collect a great deal of background information, typically more than the basics required for Standards proposals using principled design. Ideally, the authors of competing proposals would work together to produce a unified paper; failing that, we must try to reach agreement on the ranking of the most important principles, which is where Committee time is best spent. For

[1] As the language evolves, the naming scheme could include more that 26 possibilities if need be, such as doubled uppercase letters, i.e., AA–AZ, BA–BZ, CA–CZ, and so on.

[2] See https://mathworld.wolfram.com/KarnaughMap.html or https://www.cambridgeinternational.org/images/285025-topic-3.3.3-karnaugh-maps-9608-.pdf

efficiency, someone appointed by LEWG would identify and reference all the papers relevant to a given policy statement and then fairly and objectively summarize all of the proposed policy statements, their motivating principles, and their concerns. Then, this LEWG designate would collect all the motivating principles, refine them, and create a table of principles ready to be characterized and ranked.

The scoring of how well each policy statement satisfies each principle is independent of the principles' rank, so that process could proceed in parallel. When priority of the principles is established and each policy statement has been scored based on compliance with each principle, a ranked compliance table can then be analyzed and discussed in committee, and the optimal policy statement can be chosen. See "Conducting a Principled-Design-Aware Standards Meeting" in [P3004].

3.4 Recording an Auditable, Maintainable Policy

The LEWG-sponsored proposal paper for each policy statement is the obvious place to record the history and status of the current policy, including

— the problem statement
— the policy statement descriptions, motivating principles, and concerns
— the curated, refined, characterized, and ranked principles
— the compliance table corresponding to all principles and policy statements
— analysis of the compliance table
— recommendations

One of the beneficial aspects of principled design is its transparent preservation of relevant historical information; each step of the decision process is preserved in the LEWG-sponsored paper for that policy.

According to *Library Evolution Policies* [P2267R0], once a policy statement is adopted, it and the question it answers are copied into SD-9 for ready access. We propose that the paper number of the *LEWG-sponsored policy proposal* that was used to record and adopt the policy (e.g., via principled-design techniques) also be linked here. Someone who wants to understand the policy and a related decision or who wants to amend or contest it would start here. If that policy statement were to be revised, all the relevant history is naturally preserved in the next LEWG version of the policy paper.

3.5 Appealing a Policy Decision

Imagine that a policy statement has been adopted, and later a Committee member observes that the policy statement has a severe flaw that forbids or forces using a feature where doing so is clearly ill advised. Once the Committee is given an example in which the policy statement is not helpful, we can retrospectively determine where the process went wrong.

— Was a better policy statement missing from those considered?
— Was an important principle overlooked?
— Was a principle mischaracterized and hence incorrectly ranked?
— Was a policy statement inaccurately scored for a given principle?
— Has new information arisen?

Disliking a policy statement but being unable to identify a specific flaw in it is not actionable. A flaw that can be attributed to one or more procedural errors during the principled-design process is automatically basis for appeal. A proposal that focuses on the process mistakes and the recommended remedy would then be produced.

New principles and/or new policy statements are also basis for appeal. The Committee member simply adds their principles and/or policy statements to the mix, scores them, and submits that proposal for consideration. The material to be considered is the priority of any new principles, how existing policy statements are scored against them, and how any new policy statements are scored against all existing and new principles.

3.6 Road Map

This principled approach to determining an optimal policy statement for a given library question is an effective structuring and visualization tool but does not reduce the correct answer to a closed formula. Once the principled-design process is complete, the best-scoring policy statements must be considered along with the important principles they satisfy. A final choice — possibly a new alternative created with knowledge gained during the process — will need human consensus.

The remainder of this paper offers an example of what an LEWG-sponsored proposal might look like, drawn from existing papers by John Lakos and Ben Craig. We conclude with a recommendation for using principled design to select and archive LEWG policies for use of C++ language features. An appendix provides three lightly worked examples of proposals for policies currently relevant to LEWG, as though presented by individual Committee members.

4 An LEWG-Sanctioned Policy Paper for Applying the `noexcept` Specifier

In this section, we model how a representative of the LEWG might produce an objective paper on related proposed policy statements, one of which might be adopted by LEWG. We have chosen a real-world example that captures the current state of discourse regarding when to apply the `noexcept` specifier to Standard Library functions. We make an honest effort to present from the disinterested perspective of an LEWG representative, rather than from the perspective of an advocate for a particular policy.

4.1 Background and Content Preparation

C++11 was nearing release when David Abrahams discovered a serious problem with move semantics that manifests if a move constructor can throw in the presence of a function that promised the strong exception-safety guarantee.[3] At the Pittsburgh meeting, the need for some solution was discovered, and the `noexcept` operator and specifier were born solely to adorn move constructors when arbitrary types were used in a generic context that required preserving the strong guarantee. The question then became when, if ever, `noexcept` should be used otherwise.

As one might expect, with no policy, the `noexcept` operator and specifier were applied with almost no consistency or shared understanding to arbitrary functions within the Standard Library. At the final Standards meeting (Madrid, February 2011) before C++11 was released, Alisdair Meredith and John Lakos — two authors of this paper — proposed a stopgap policy whose centerpiece was to prevent putting the `noexcept` specifier on functions having narrow contracts (in anticipation of a Contracts facility that would turn out to be surprisingly elusive[4]). At the time, using the feature where it *appeared* to do no harm garnered much enthusiasm, so a set of hastily prepared rules — including the Lakos Rule – were agreed to with a consensus of roughly 85%[5]:

a. No library destructor should throw. They shall use the implicitly supplied (non-throwing) exception specification.
b. Each library function having a wide contract (i.e., does not specify undefined behavior due to a precondition) that the [LEWG and] LWG agree cannot throw, should be marked as unconditionally `noexcept`.
c. If a library swap function, move-constructor, or move-assignment operator is conditionally-wide (i.e. can be proven to not throw by applying the `noexcept` operator) then it should be marked as conditionally `noexcept`.
d. If a library type has wrapping semantics to transparently provide the same behavior as the underlying type, then default constructor, copy constructor, and copy-assignment operator should be marked as conditionally `noexcept` [so that] the underlying exception specification still holds.
e. No other function should use a conditional `noexcept` specification.
f. Library functions designed for compatibility with "C" code (such as the atomics facility), may be marked as unconditionally `noexcept`.

The greatest controversy centers around the clause in b that precludes putting `noexcept` on a narrow contract. If we remove that clause from b, we have

> b´. Each library function that the [LEWG and] LWG agree cannot throw should be marked as unconditionally `noexcept`.

Ben Craig has suggested[6] his preferred wording, which we agree is equivalent.

> b´´. Each library function that will not throw when called with all preconditions satisfied should be marked as unconditionally `noexcept`.

[3] See [N2855].
[4] Note that an MVP for a solid Contracts facility — created with substantial and effective use of principled design [P3004] — is finally on track for C++26; see [**P2900R4?**].
[5] Copied verbatim from [N3279], updated later by [P0884R0].
[6] See [P3085R0].

4.1.1 Decisions Needed

A number of decisions must be made to identify the universe of potential policy directions to be considered. We now describe those various decision points and then consider which combinations of decisions would lead to coarsely specified *directional* policy statements worth considering in greater detail. Each component part of a policy design will be described and assigned a short (all capital) identifier, and later we will build complete policy designs out of these parts.

4.1.1.1 Decision 1: Minimize or Maximize `noexcept` (MIN or MAX)

The first question to consider is whether to continue to maximize or to instead minimize the set of functions to be considered for marking as `noexcept`. One choice is to place `noexcept` on any function that will not throw when invoked in contract, thus maximizing it.

> MAX (Maximum `noexcept`) — Any function that does not throw in contract shall be marked as `noexcept`. Put another way, any function that would be documented as "throws nothing" — and thus indicates that those who wrote and reviewed the specification (LEWG and LWG) have agreed no implementation of this function may ever throw in contract — shall be marked `noexcept`.

Since the introduction of `noexcept` and the adoption of policies on the use of `noexcept` in the Standard library, much has been learned about the pros and cons of having `noexcept` used liberally in practice.

- The presence of the `noexcept` operator often affects code generation: It frequently (but not always) reduces the size of the object code but, contrary to popular belief, does *not* improve average run time on modern platforms.
- An aggressive shift within the greater programming community toward safety (security in particular) has forced WG21 to move aggressively toward making C++ a safer language in terms of both security and correctness. A prime example of improving both security and correctness was achieved with the introduction of *erroneous behavior* to replace what was previously UB with well-defined behavior that is still treated as a defect. Perhaps the most promising way of avoiding UB in C++ is the widespread use of SG21's burgeoning Contracts MVP, which is now essentially feature complete and on track for a C++26 release. A concern is that — even with a throwing handler — detection of a contract violation will not allow a typical program to recover from an error instead of terminate because `noexcept` specifiers are sprinkled throughout both Standard and user code. As a result, a prodigious effort has been made recently to reduce the use of `noexcept` to only truly necessary situations, i.e., when an algorithmic performance gain is to be had by querying, from a generic context, whether the function has a nonthrowing exception specification.

Nearly 15 years of experience with `noexcept` has revealed that applying `noexcept` superfluously in the Library can be highly detrimental to writing safe and reliable software. Therefore, the committee must scrupulously reconsider when `noexcept` is truly needed, which is when a function must — not just theoretically but in practice — have the `noexcept` operator applied to it (i.e., in a generic context) to enable an algorithmically better approach to be taken by a templated function. Up to now, only move construction and move assignment are used in this way by generic functions in the Standard Library itself, while `swap` is similar enough to potentially be useful in algorithms from other libraries. No other functions in any library have ever been known to demonstrate such value. Hence, until an existence proof can be demonstrated to LEWG, the argument can be made that no other functions shall be candidates for being marked as unconditionally `noexcept`.

> MIN (Minimum `noexcept`) — Only those functions with a sanctioned need (e.g., practical application of the `noexcept` operator) to be marked `noexcept` shall be marked unconditionally `noexcept`. Specifically, move constructors, move assignment operators, and the `swap` function have all been shown to be used in algorithms where significant algorithmic improvements can be made when these functions can be determined, at compile time, to not throw. No other functions shall be marked `noexcept` absent an update to this policy.

Adopting a policy statement incorporating this decision would include adopting this definition of what functions *need* to be marked as `noexcept` along with the implicit meta-policy that a general consensus for a need must be achieved before additional kinds of functions may be added to this list.

4.1.1.2 Decision 2: Apply the Lakos Rule (with Potential Exemptions) (LAK, NLM, NLC)

The second decision to be made is whether the Lakos Rule should also be applied.

> LAK (Lakos Rule) — Each library function having a narrow contract (i.e., one that has any form of precondition) shall not be marked `noexcept`.

The Lakos Rule has two known exemptions to consider. The first is largely John Lakos' own fault for popularizing the use of local arena memory allocators. Both move construction and move assignment will be leveraged by many algorithms only when they are also marked `noexcept`; if these operations potentially throw, algorithms must instead fall back to making copies to provide the strong exception-safety guarantee. Otherwise, move operations on allocator-aware objects must either themselves fall back to a potentially-throwing copy or have a precondition requiring matching allocators. Clearly, in the presence of the Lakos Rule, these operations can never be marked `noexcept` for allocator-aware types.

> NLM (No Lakos Rule on Move) — A move operation that will not throw in contract may always be marked as unconditionally `noexcept`, even when it has a narrow contract.

When `noexcept` was first introduced into the C++ language, the extent of its impact on code generation and performance was unclear. That a C++ library function, such as those in the `<atomics>` API, intended to mirror a C library function could never throw was perfectly obvious. Hence, the original LEWG policy on the use of `noexcept` included an exemption for library functions designed for compatibility with C code.

> NLC (No Lakos Rule on C Compatibility) — A library function designed for compatibility with C code that does not throw in contract may always be marked as unconditionally `noexcept`, even when it has a narrow contract.

4.1.1.3 Decision 3: Remove Permission for Implementations to Strengthen Exception Specifications (POR)

Since C++03, implementations have been permitted to strengthen the exception specification of a Standard Library function over that which is specified in the C++ Standard. While generally unobservable like an attribute, the `noexcept` operator in C++11 allowed applications to query for the exception specification of a function and alter control flow based on this property. Since C++17, the exception specification of a function has been extended to become part of the type system, resulting in an even larger impact on the ability to substitute one declaration for another.

To allow an implementation to strengthen an exception specification is to allow that implementation to unilaterally change that function's type (for all arguments) and to allow the `noexcept` operator to take different code paths on different platforms. Some feel strongly that such flexibility is inconsistent with language design imperatives because it can reduce the predictability of program behavior when porting from one platform to another. Note that no such permission is granted for other language features, such as `constexpr`, `explicit`, or `const`.[7]

> POR (Portable) — To maximize portability, Standard Library implementations may no longer strengthen the exception specification of a function beyond that which is delineated in the Standard.

4.1.1.4 Decision 4: Allow Throwing Destructors (NTD)

Types having throwing destructors are highly problematic since an exception thrown from a destructor during stack unwinding will generally result in having two exceptions in flight and an inevitable invocation of `std::terminate` even if the destructor is not marked `noexcept`. Therefore, the Standard Library chooses never to deviate from the implicit *non*throwing exception specification on destructors. We have seen no good motivation to deviate from this policy.

[7] To be fair, clients' applying the `noexcept` operator to the invocation of any function in the C++ Standard Library is possible (thereby making the removal of `noexcept` a breaking change in theory); though we see no practical purpose for real-world applications doing so, see *Hyrum's Law*: https://www.hyrumslaw.com/.

NTD (No Throwing Destructors) — No library destructor shall throw; they shall use the implicitly supplied (nonthrowing) exception specification.

4.1.1.5 Decision 5: Allow Conditional `noexcept` (CNN)

The purpose and effect of *conditional* `noexcept` is manifestly distinct from that of *unconditional* `noexcept`. In general, conditional `noexcept` facilitates writing generic code where the decision to use `noexcept` is being made by another type, and thus the responsibility of a generic type is to propagate that decision where practicable. Therefore, we see no reason to consider changes to the previous policies pertaining to conditional `noexcept`, and the three clauses that already exist cover the use of that distinct language feature. (Note that these rules apply also to function, constructor, and member-function templates.)

CN1 — If a library `swap` *function*, *move constructor*, or *move-assignment operator* is conditionally nonthrowing (i.e., can be proven to not throw by applying the `noexcept` operator), then that function, constructor, or member function shall be marked as conditionally `noexcept`.

CN2 — If a library type has *wrapping* (a.k.a. *proxy*) semantics that are intended to transparently provide the same behavior as the underlying type, then *default constructor*, *copy constructor*, and *copy-assignment operator* shall be marked as conditionally `noexcept`, thereby allowing the underlying exception specification to be exposed.

CN3 — No other function, constructor, or member function shall have a conditional `noexcept` specification.

CNN — Use all three policies: CN1 + CN2 + CN3.

4.1.1.6 Decision 6: Minimal `noexcept` Includes Asynchronous Customization Points (NCP)

We see one more potential exemption to this minimum family of policies. When employing any variant of the Min `noexcept` policy so far, the only valid reason to place `noexcept` on a Standard Library function is if a reasonable use case can be exhibited wherein the `noexcept` operator can be used in a generic context to write code that is potentially more runtime efficient. The only known case where that need exists is operations for which the strong exception-safety guarantee is required.

Much more recently, however, a completely different application of the `noexcept` specifier has emerged, this time applied to certain customization point functions when integrating with asynchronous execution frameworks, such as sender/receiver.

A second potential use case occurs when a generic framework requires callbacks that are declared `noexcept` in an attempt to ensure that the function passed in does not throw. Some might consider this use case dubious because, if a function doesn't throw in contract, it can easily be wrapped with a generic `noexcept` wrapper function that will terminate if an exception is thrown.

Other ways are available to make such a framework behave as desired (e.g., a default handler that, if an exception is thrown, terminates) that would not force potential clients to adulterate their own functions with excessive use of `noexcept`, thereby removing their ability to effectively test their own narrow contracts using a throwing contract-checking handler. Allowing this exception in the Standard would in effect force users to wrap their functions for a special use and/or hide that their functions might throw if called out of contract with a wrapper that will just terminate.

NCP (`noexcept` on Customization Points) — Allow `noexcept` on C++ functions that are customization points that will be used with asynchronous frameworks that have conditional support for handling potentially throwing functions.

This special case would also apply to taking the addresses of such functions as function parameters, in which case `noexcept` would appear as part of the type of one or more parameters of such function.

4.1.2 Viable Policy Statements

We'll now combine the various potential decisions elucidated in the previous section to identify potentially viable policy-statement directions worth considering. Although, in more general principled design, we would typically

wait to assign capital-letter labels until just before the compliance table, our use of supporting tables and Karnaugh maps in this section make doing so here desirable. Note that we will start with Policy Statement B because we reserve Policy Statement A for the no-policy option, which in this case is also the status quo.[8]

Each of these options proposes the same answers for the last two questions posed above, namely that we retain the current policies for the use of conditional `noexcept` (CNN) and a nonthrowing destructor (NTD). Hence, each of the polices below should be assumed to have appended those sections of the current policy.

— Policy Statement B — Basic Maximal `noexcept` (MAX)

In this analysis, we take our first candidate policy (MIN) from the original 2011 policy and modify the wording in b to remove the Lakos Rule (LAK). Note that since 2019, several proposals have explored Policy Statement B ([P1656R2], [P3085R0]).

— Policy Statement C — Maximal `noexcept` with Lakos Rule and C Exemption (MAX+LAK+NLC)

The second candidate to consider is the `noexcept` policy as originally proposed and adopted back in 2011. This is the maximal rule (MAX) along the Lakos Rule (LAK) and also includes the first of the two exemptions (NLC) for libraries intended to be compatible with C. This `noexcept` policy statement is the current state since 2011 ([N3279]).

— Policy Statement D — Minimal `noexcept` with Lakos Rule (MIN+LAK)

The third candidate to consider is a dramatic departure from the status quo, applying `noexcept` minimally (MIN) along with the Lakos Rule (LAK). Importantly, implementations retain the freedom to strengthen exception specifications and service the needs of their clients as they see fit. Hence, adopting this policy need not impact any current implementations or their clients. Note that here we do not need to grant the exception to the Lakos Rule for C compatible functions (NLC) because we would no longer be expected to put `noexcept` on them in the first place (but implementations would still be free to do so under this proposal).

— Policy Statement E — Minimal `noexcept` (MIN)

This fourth candidate, which is effectively Policy D without the Lakos Rule, might seem ill-advised given the history of arguments advocating for it. In particular, given that the guidance of MIN is to use `noexcept` only when *needed*, having the Lakos rule applied on top of MIN introduces friction that, on balance, is unwarranted. That is, if a valid need or concern exists, then perhaps safety will need to give way to performance in those rare situations. If all cases of strong need would lead to exemptions to the Lakos Rule, then we would be wise to consider using `noexcept` minimally (MIN) without considering the Lakos Rule (LAK) at all.[9]

— Policy Statements F and G — Nonstrengthening D and E (MIN+LAK+POR, MIN+POR)

Next, for the restrictive `noexcept` policies where we seek to minimize the use of `noexcept` in the Standard Library, we should also consider adopting the portability policy removing the ability for implementations to strengthen the exception specification of the functions they implement.

— Policy H — Minimal `noexcept` with Customization Point Allowances (MIN+NCP)

Finally, we consider extending our definition of *minimal* to include functions that will be used as callbacks for asynchronous operations.

[8] Informally, the status quo is somewhere between B and C (but much closer to C) below.

[9] Note that, as things stand today, MIN and MIN+LAK+NLM — i.e., the Minimal `noexcept` policy with the addition of the Lakos Rule along with its first exemption — are equivalent in practice.

In summary, we can view the full space of solutions we intend to consider in a single table.

S	MIN	LAK	POR	NCP	Comments
B					[P3085R0], Ben Craig's policy statement proposal
C		+NLC			[N3279] and [P0884R0], original `noexcept` policy statement
D	MIN	LAK			Minimal version of original policy statement
E	MIN				Minimal use of `noexcept` policy statement
F	MIN	LAK	POR		Restrictive and minimal version of original policy statement
G	MIN		POR		Most minimal and restrictive policy statement
H	MIN			NCP	Minimal version w/allowance for modern evolution policy statement

We can also view these various policy statements using multidimensional Karnaugh maps to understand how the basic properties vary. Consider first this two-dimensional Karnaugh map showing the first four policy statement candidates, which differ based on the first two decisions.

```
<-- MAX --x-- MIN -->
+----+----+----+----+
| B  | C+ | D  | E  |
+----+----+----+----+
     <-- LAK -->
```

As we can see, policy statement B is MAX without LAK, C is MAX with LAK (+ the first exception), D is MIN with LAK (no exceptions), and E is MIN without LAK.

Now that we have the basic directions, we can take a look at the largely orthogonal decision of whether to remove permission for implementations to strengthen the exception specifications stated in the specification of the C++ Standard Library (POR). Visualizing all our candidate solutions so far can be done using a three-dimensional Karnaugh map.

```
<-- MAX --x-- MIN -->
+----+----+----+----+  ^
|    |    | F  | G  | POR
+----+----+----+----+  v
| B  | C+ | D  | E  |
+----+----+----+----+
     <-- LAK -->
```

Notice that we have not considered, as viable policy statements, removing implementations' ability to strengthen explicit `noexcept` specification in the Standard (POR) under the maximum `noexcept` approach (MAX), with or without the Lakos rule (LAK). In a world where the Standard Library heavily incorporates use of the `noexcept` specifier, a particular implementation providing even more such annotations where appropriate would likely be welcomed. Hence, the decision to address only a subset of the full space of policies is guided by what one would reasonably consider viable solutions. If a solution that has a dearth of adjacent considered solutions gains consensus, further analysis will be required in that direction.

Now that we have provided the information we feel is necessary to identify the various design alternatives, we start the principled-design process to help us distinguish which of the identified-as-viable policy statements best satisfy the most important principles that motivate the need for this policy.[10]

4.2 Problem Statement

Provide a policy statement that describes when a Standard Library function is allowed and/or required to be marked as being `noexcept` as well as if and how an implementation is permitted to deviate from that specification.

[10] Note that determining an optimal statement for a policy is a particular application of the general principled-design process described in [P3004].

4.3 Probative Questions

— How important is minimizing program size?
— How important is minimizing UB?

4.4 Policy Statements

Each potential policy statement follows, including

— a succinct identifier (single capital letter) unambiguously referencing the corresponding design choice in the background
— a title
— a description of the composition of parts, e.g., (MAX+LAK+NLC)
— a concise, plain-language description sufficient to distinguish this candidate policy statement from all others
— TBD: formal wording for this policy statement should it be adopted
— Motivating Principles
  — Each principle is concise and favors this policy statement choice.
    — Additional clarifying information may be added in bullets below.
— Concerns
  — Each concern describes a potential shortcoming with this policy statement choice.
    — Additional related information may be added in bullets below.

4.4.1 B: Basic Maximal `noexcept` (MAX)

— Short Description
  — Basic maximal use of `noexcept` without the Lakos Rule [P3085R0]
— Formal Statement Wording
  — TBD
— Motivating Principles
  — Enable algorithmic runtime optimizations.
    — All functions that do not throw in contract are so marked.
  — Enable use of `noexcept` to restrict function arguments (e.g., callbacks).
    — There are no restrictions beyond not throwing in contract.
  — Minimize object-code size.
    — Functions declared `noexcept` tend to be smaller than those that
    — aren't.[11]
  — Minimize executable size.
    — Not all object code necessarily moves into the executable, e.g., instantiated inline functions and function templates.
  — Express defined behavior directly in code.
    — This is similar in spirit to `constexpr`, `explicit`, and `const`.
— Concerns
  — Severely impedes use of a throwing handler for out-of-contract calls
  — Impedes backward-compatible extensions of standard functions
    — Breaks Liskov Substitutability for widening narrow contracts to include throwing
  — Severely impedes use of Contracts for handling defects (even if just temporarily)
    — A throwing violation handler will fail to recover and instead terminate if a bug is detected in any function marked `noexcept`.
  — Prevents efficient portable negative testing
    — Requires the use of inefficient death tests and specific frameworks (e.g., gtest)
  — Average runtime performance is not meaningfully improved by marking it `noexcept`.
    — The overwhelming preponderance of evidence suggests no reliable difference in performance (either positive or negative) occurs outside of a few degenerate microbenchmarks.[12]

[11] lakos20, see `noexcept` Specifier, Reducing object-code size, pp. 1101-1111
[12] lakos20, see `noexcept` Specifier, Unattainable runtime performance benefits, pp. 1134-1143

4.4.2 C: Maximal `noexcept` with Lakos Rule and C Exemption (MAX+LAK+NLC)

- Short Description
  - Original maximal use of `noexcept` along with Lakos Rule and an exemption for functions intended to interoperate with C [N3279]
    - Same as B above but adds the Lakos Rule with a C-language exemption
- Formal Statement Wording
  - TBD
- Motivating Principles
  - Allow implementers to choose to support throwing contract-violation handlers.
    - This feature has been determined to be essential for the Contracts MVP.
  - Allow implementations to enable users to achieve safe shutdown in production.
    - An implementation can now define out-of-contract calls to throw.
  - Allow implementers to enable handling and continuing in production.
    - An implementation can now define out-of-contract calls to throw.
  - Enable effective, portable negative testing for standard functions.
    - Does not require frameworks like gtest that support death tests
  - Provide a good model (e.g., w.r.t. safety) for retail users.
    - Best practice for typical users arguably favors safety over code size.
- Concerns
  - The abundance of exceptions make safe handling in production dubious.
    - We cannot handle and quit reliably.
    - We cannot handle and recover reliably.
  - Note that there is no concern for direct negative unit testing.
    - The Lakos Rule ensures that exceptions can be used for effective and portable unit testing.

4.4.3 D: Minimal `noexcept` with Lakos Rule (MIN+LAK)

- Short Description
  - Minimal use of `noexcept` and the Lakos Rule (no exemptions needed)
    - Same as C above but employs *minimal* rather than *maximal* use of `noexcept`
- Formal Statement Wording
  - TBD
- Motivating Principles
  - Maximize an implementation's flexibility to add `noexcept`
    - Maximizes freedom to strengthen exception specifications
  - Minimize the chance that a propagated exception will cause unintended termination.
    - Minimizes use of `noexcept` where not needed for algorithmic optimization.
  - Accommodate the multiverse.
    - Minimizes the extent to which effective use of language features will be suppressed by the C++ Standard in favor of particular uses.
  - Provide a great model (e.g., w.r.t. safety) for retail users.
    - Best practice for typical users arguably favors safety over code size.
- Concerns
  - This policy favors avoiding the use of `noexcept` on narrow contracts.
    - There is one known case where this restriction is an issue: a move assignment where the allocators do not compare equal.
    - The common use for performance outweighs the ability to test and recover easily.
    - Again, this use of `noexcept`on the narrow-contract move assignment is an absolute edge case, but one that must be considered.

4.4.4 E: Basic Minimal `noexcept` (MIN)

- Short Description
  - Basic minimal use of `noexcept`

        - Same as D above but without the Lakos Rule
- Formal Statement Wording
    - TBD
- Motivating Principles
    - Allow for narrow contracts that need to be `noexcept`.
        - E.g., this allows for move assignment with unequal allocators to be `noexcept`.
    - Provide an ideal model (e.g., w.r.t. safety or performance) for retail users.
        - Best practice arguably favors safety over code size but not performance.
- Concerns
    - Some people will feel that Lakos sold out, but not so.
        - This policy statement puts the priority where it belongs.
    - This and all previous solutions allow for vendors to add `noexcept`.
        - `noexcept` (since C++17) is part of the type system; hence, it is not an implementation detail.
        - If some vendors add `noexcept` and others don't, we get portability issues, including the potential to take separate code paths.
        - Such different code paths are unlikely to occur in practice because there's no need to test such functions using the `noexcept` operator.
        - Then again, given Hyrum's Law, perhaps some will attempt to do so after all.

4.4.5 F: Minimal `noexcept` with Lakos Rule but No Strengthening (MIN+LAK+POR)

- Short Description
    - Minimal use of `noexcept` along with the Lakos Rule but forbidding implementers to strengthen exception specifications delineated in the specification of the C++ Standard Library
        - Same as D above but removes permission for implementors to strengthen exception specifications
- Formal Statement Wording
    - TBD
- Motivating Principles
    - Ensure that standard functions have the same meaning across conforming implementations.
        - I.e., all standard functions have the same type and thus respond in identical ways to all reflection type queries (e.g., the `noexcept` operator).
    - Maximize the portability of code written using the Standard Library.
        - Permission to add `noexcept` changes the type of the function.
- Concerns
    - Fails to address the move assignment issue by denying permission allowing the narrow-contract move assignment to be marked as `noexcept`
    - Potentially forces ABI breaks or deviation from the Standard
        - Specifically, implementations that currently have `noexcept` on functions that are not so marked in the Standard
        - Of particular concern are functions whose parameters are of function type that have been strengthened and any function templates.
    - Removes ability for implementers to selectively reduce code size
        - E.g., to override the Lakos Rule or to strengthen based on a proprietary implementation

4.4.6 G: Basic Minimal `noexcept` but No Strengthening (MIN+POR)

- Short Description
    - Basic minimal use of `noexcept` but forbidding implementers to strengthen exception specifications delineated in the specification of the C++ Standard Library
        - Same as E above but removes permission for implementors to strengthen exception specifications
- Formal Statement Wording
    - TBD
- Motivating Principles
    - Ensure that standard functions have the same meaning across conforming implementations.
    - Maximize the portability of code written using the Standard Library.

    - Allow for narrow contracts that need to be `noexcept`.
- Concerns
    - Potentially forces ABI breaks or deviations from the Standard
    - Removes ability for implementers to selectively reduce code size

4.4.7 H: Basic Minimal `noexcept` and Function Parameter Exception (MIN+NCP)

- Short Description
    - Minimal use of `noexcept` but giving permission for certain framework functions to restrict their callback functions by declaring these function parameters to be `noexcept`
        - Same as E above but makes an exception that would allow frameworks to restrict (at compile time) callbacks to be non-throwing
- Formal Statement Wording
    - TBD
- Motivating Principles
    - Allow `noexcept` on customization point functions that will be used in asynchronous contexts.
        - In particular, functions that participate in the sender/receiver framework, despite there being no anticipated use of the `noexcept` operator to improve algorithmic performance
    - Allow `noexcept` on callback parameters to restrict argument types.
        - In particular, allow the Standard to employ use of the `noexcept` specifier for the sole purpose of restricting (at compile time) the types of callbacks that can be supplied to such frameworks.
- Concerns
    - This exception allows an arguably gratuitous use of `noexcept`.
        - There are other, better ways to achieve this goal.
        - The Standard would always handle the exception branch by default.
        - Not all errors need to be part of the type system.

4.5 Curated, Refined, Characterized, and Ranked Principles

At this point we turn our attention to assembling a complete set of relevant principles to be applied across all the candidate policy statements. Note that we would normally do all the steps below in place, but we have elected to show each step in a separate subsection to make the result of each curation step more readily comparable to the principles previous state.

4.5.1 Copying the Motivating Principles in Order

The first step is simply to copy all the motivating principles in order. Reasons and clarifications associated with a principle are not retained.

4.5.1.1 B: Basic Maximal `noexcept` (MAX)

- Enable algorithmic runtime optimizations.
- Enable use of `noexcept` to restrict function arguments (e.g., callbacks).
- Minimize object-code size.
- Minimize executable size.
- Express defined behavior directly in code.

4.5.1.2 C: Maximal `noexcept` with Lakos Rule and C Exemption (MAX+LAK+NLC)

- Allow implementers to choose to support throwing contract-violation handlers.
- Allow implementations to enable users to achieve safe shutdown in production.
- Allow implementers to enable handling and continuing in production.
- Enable effective, portable negative testing for standard functions.
- Provide a good model (e.g., w.r.t. safety) for retail users.

4.5.1.3 D: Minimal `noexcept` with Lakos Rule (MIN+LAK)

— Maximize an implementation's flexibility to add `noexcept`.
— Minimize the chance that a propagated exception will cause unintended termination.
— Accommodate the multiverse.
— ~~Provide a great model (e.g., w.r.t. safety) for retail users.~~

4.5.1.4 E: Basic Minimal `noexcept` (MIN)

— Allow for narrow contracts that need to be `noexcept`.
— ~~Provide an ideal model (e.g., w.r.t. safety/performance) for retail users.~~

4.5.2 F: Minimal `noexcept` with Lakos Rule but No Strengthening (MIN+LAK+POR)

— Ensure that standard functions have the same meaning across conforming implementations.
— Maximize the portability of code written using the Standard Library.

4.5.3 G: Basic Minimal `noexcept` but No Strengthening (MIN+POR)

— ~~Ensure that standard functions have the same meaning across conforming implementations.~~
— ~~Allow for narrow contracts that need to be `noexcept`.~~
— ~~Maximize the portability of code written using the Standard Library.~~

4.5.4 H: Basic Minimal `noexcept` and Function Parameter Exception (MIN+NCP)

— Allow `noexcept` on customization point functions that will be used in asynchronous contexts.
— ~~Allow `noexcept` on callback parameters to restrict argument types.~~

4.5.5 Removing Duplicates

The next step is simply to look for principles that are, for scoring purposes, equivalent to a previous one and simply delete the latter one in place. Note that we have indicated removed principles via strikethrough in the previous subsection.

4.5.6 Normalizing Wording

4.5.6.1 B: Basic Maximal `noexcept` (MAX)

— Maximize algorithmic runtime optimizations.
— Maximize use of `noexcept` to qualify function parameters.
— Minimize object-code size.
— Minimize executable size.
— Maximize expressing defined behavior directly in code.

4.5.6.2 C: Maximal `noexcept` with Lakos Rule and C Exemption (MAX+LAK+NLC)

— Maximize implementations' ability to support throwing contract-violation handlers.
— Maximize implementations' ability to enable users to achieve safe shutdown in production.
— Maximize implementations' ability to enable users to handle and continue in production.
— Maximize effective, portable negative unit testing for standard functions.
— Maximize a desirable best-practices model (safety/performance) for typical retail users.

4.5.6.3 D: Minimal `noexcept` with Lakos Rule (MIN+LAK)

— Maximize an implementation's flexibility to add `noexcept`.
— Minimize the chance that a propagated exception will cause unintended termination.
— Minimize the likelihood of disenfranchising a member of the multiverse.

4.5.6.4 E: Basic Minimal `noexcept` (MIN)

— Maximize ability to use `noexcept` for narrow contracts having algorithmic need.

4.5.6.5 F: Minimal `noexcept` with Lakos Rule but No Strengthening (MIN+LAK+POR)

— Minimize divergence in meaning of standard functions across conforming implementations.
— Maximize portability for code written using the Standard Library.

4.5.6.6 G: Basic Minimal `noexcept` but No Strengthening (MIN+POR)

4.5.6.7 H: Basic Minimal `noexcept` and Function Parameter Exception (MIN+NCP)

— Allow `noexcept` on customization-point functions used in asynchronous contexts.

4.5.7 Splitting Principles (as Needed)

At this point, we consider whether some principles would be better if split into two separate principles. For example,

— Maximize performance.

might be better represented as

— Minimize compile time.
— Minimize run time.

or

— Maximize backward compatibility.

could be split into

— Maximize backward compatibility with respect to compilation.
— Maximize backward compatibility with respect to defined runtime behavior.

In this case, however, we have nothing we feel we need to split.

4.5.8 Merging Principles (as Needed)

If, in the process of splitting, we found that we could now merge one or more parts into another principle, then now would be the time to do that. As it turns out, there was no splitting, and we already removed what we considered duplicates (e.g., the use of `noexcept` to restrict function-pointer parameters, as called out with slightly different wording in both B and H).

4.5.9 Adding New Relevant Principles (as Needed)

If we, as the LEWG-sponsored curators of the principled-design approach to policy selection, discover additional principles — general or specific — that might be relevant to the decision, then we would be wise to list, characterize, and score them, even if we're not sure of their relevance or importance. In that spirit, we list a few more potential principles here.

— Minimize the amount of work needed to bring all Standard Libraries in line with the current policies.
— Minimize the number of discrepancies that will or must remain between the Standard and the current policies.
— Minimize the cost for implementers to remain conformant while meeting the demands of their customers.

We then refine our original thoughts into more appropriate wording.

— Minimize effort to align Standard Libraries with this policy statement.
— Minimize likely persistent discrepancies between the Standard and this policy statement.
— Minimize effort for implementations to remain conformant while satisfying their client base.

Again, we note that how these principles are scored relative to the more specific ones is a topic to be addressed within a meeting of the LEWG.

4.5.10 Creating Principle IDs

The next step is to provide short mnemonic principle IDs.

Principle ID	Principle Statement
AlgoOpt	Maximize algorithmic runtime optimizations.
ArgRestrict	Maximize use of `noexcept` to qualify function parameters.
MinObjSize	Minimize object-code size.
MinExeSize	Minimize executable size.
MaxExpressInCode	Maximize expressing defined behavior directly in code.
MaxImplFlexHand	Maximize implementations' ability to support throwing contract-violation handlers.
MaxUserSafeShut	Maximize implementations' ability to enable users to achieve safe shutdown in production.
MaxUserSafeRecov	Maximize implementations' ability to enable users to handle and continuing in production.
MaxNegUnitTest	Maximize effective, portable negative unit testing for standard functions.
MaxBestPracExamp	Maximize a desirable best-practices model (safety/performance) for typical retail users.
MaxImpFlexAddNoex	Maximize an implementation's flexibility to add `noexcept`.
MinUnintendedExit	Minimize the chance that a propagated exception will cause unintended termination.
MaxMultiverse	Minimize the likelihood of disenfranchising a member of the multiverse.
MaxUseForAlgoNeed	Maximize ability to use `noexcept` for narrow contracts having algorithmic need.
MinDivergMeaning	Minimize divergence in meaning of standard functions across conforming implementations.
MaxUserPortabilty	Maximize portability for user code written using the Standard Library.
AllowOnAsyncFrame	Allow `noexcept` on customization-point functions used in asynchronous contexts.
MinCostToFixStd	Minimize effort to align the current Standard Library with this policy statement.
MinPermDiverge	Minimize likely persistent discrepancies between the Standard and this policy statement.
MinCostToImpls	Minimize effort for implementations to remain conformant while satisfying their client base.

4.6 Characterizing and Ranking the Principles

We earnestly attempt to fairly score the principles for importance and objectivity. All of these ratings are, of course, tentative and subject to review by authors of competing papers and other interested members of LEWG and beyond. The proposal is merely a starting point, to save time and to focus the discussion on points of disagreement.

Rank	i	o	Principle ID	Principle Statement
1	9	@	AlgoOpt	Maximize algorithmic runtime optimizations.
14	1	@	ArgRestrict	Maximize use of `noexcept` to qualify function parameters.
20	-	@	MinObjSize	Minimize object-code size.
12	1	@	MinExeSize	Minimize executable size.
16	1	5	MaxExpressInCode	Maximize expressing defined behavior directly in code.
2	9	5	MaxImplFlexHand	Maximize implementations' ability to support throwing contract-violation handlers.
5	9	5	MaxUserSafeShut	Maximize implementations' ability to enable users to achieve safe shutdown in production.

Rank	i	o	Principle ID	Principle Statement
6	9	5	MaxUserSafeRecov	Maximize implementations' ability to enable users to handle and continuing in production.
17	1	5	MaxNegUnitTest	Maximize effective, portable negative unit testing for standard functions.
11	5	-	MaxBestPracExamp	Maximize a desirable best-practices model (safety/performance) for typical retail users.
15	1	@	MaxImpFlexAddNoex	Maximize an implementation's flexibility to add `noexcept`.
4	9	5	MinUnintendedExit	Minimize the chance that a propagated exception will cause unintended termination.
7	9	5	MaxMultiverse	Minimize the likelihood of disenfranchising a practicing member of the multiverse.
3	9	5	MaxUseForAlgoNeed	Maximize ability to use `noexcept` for narrow contracts having algorithmic need.
8	5	@	MinDivergMeaning	Minimize divergence in meaning of standard functions across conforming implementations.
9	5	5	MaxUserPortabilty	Maximize portability for user code written using the Standard Library.
13	1	@	AllowOnAsyncFrame	Allow `noexcept` on customization-point functions used in asynchronous contexts.
18	1	-	MinCostToFixStd	Minimize effort to align the current standard library with this policy statement.
19	1	-	MinPermDiverge	Minimize likely persistent discrepancies between the Standard and this policy statement.
10	5	-	MinCostToImpls	Minimize effort for implementations to remain conformant while satisfying their client base.

Note that in all candidate policy statements, object size tracks with executable size and cannot further discriminate between policy statements in the compliance table. Hence, we have scored the importance level of MinObjSize as irrelevant (-), and we do not further score this principle in the compliance table.

4.7 The Compliance Table: Solutions Scored Against Ordered Principles

A: The no-policy statement (currently status quo) (no policy) B: Basic Maximal `noexcept` (MAX) C: Maximal `noexcept` with Lakos Rule and C Exemption (MAX+LAK+NLC) D: Minimal `noexcept` with Lakos Rule (MIN+LAK) E: Basic Minimal `noexcept` (MIN) F: Minimal `noexcept` with Lakos Rule but No Strengthening (MIN+LAK+POR) G: Basic Minimal `noexcept` but No Strengthening (MIN+POR) H: Basic Minimal `noexcept` and Function Parameter Exception (MIN+NCP)

Rank	i	o	Principle ID	A	B	C	D	E	F	G	H
1	9	@	AlgoOpt	3	9	7	7	9	7	9	9
2	9	5	MaxImplFlexHand	3	1	7	7	9	7	9	9
3	9	5	MaxUseForAlgoNeed	3	1	7	7	9	7	9	9
4	9	5	MinUnintendedExit	3	1	3	7	9	7	9	9
5	9	5	MaxUserSafeShut	3	1	3	7	9	7	9	9
6	9	5	MaxUserSafeRecov	3	1	3	7	9	7	9	9
7	9	5	MaxMultiverse	3	1	5	7	9	7	9	9
8	5	@	MinDivergMeaning	5	7	7	3	3	9	9	3
9	5	5	MaxUserPortabilty	5	7	7	3	3	9	9	3
10	5	-	MinCostToImpls	@	9	7	7	9	1	1	9
11	5	-	MaxBestPracExamp	1	3	5	7	9	7	9	8
12	1	@	MinExeSize	5	9	8	1	3	1	3	4

Rank	i	o	Principle ID	A	B	C	D	E	F	G	H
13	1	@	AllowOnAsyncFrame	@	@	7	-	-	-	-	@
14	1	@	ArgRestrict	@	@	@	-	-	-	-	@
15	1	@	MaxImpFlexAddNoex	5	2	1	8	9	-	-	9
16	1	5	MaxExpressInCode	5	9	8	2	1	2	1	1
17	1	5	MaxNegUnitTest	5	1	9	@	9	@	9	9
18	1	-	MinCostToFixStd	@	8	9	7	6	6	5	5
19	1	-	MinPermDiverge	@	9	8	8	9	8	9	@
20	-	@	MinObjSize								

4.8 Analysis of the Compliance Table

Normally we would row-by-row examining each candidate policy statement's column to see which, if any, are eliminated. A quick inspection of the first seven rows shows that candidate policy statements E, G, and H are all tied for what is effectively the maximum score. We will, therefore, ignore the other columns and look at only these moving forward.

E: Basic Minimal `noexcept` (MIN) G: Basic Minimal `noexcept` but No Strengthening (MIN+POR) H: Basic Minimal `noexcept` and Function Parameter Exception (MIN+NCP)

Rank	i	o	Principle ID	E	G	H
8	5	@	MinDivergMeaning	3	9	3
9	5	5	MaxUserPortabilty	3	9	3
10	5	-	MinCostToImpls	9	1	9
11	5	-	MaxBestPracExamp	9	9	8

Looking at the next block of rows (with importance score of 5), the weighting clearly favors G, with E as possible alternate, depending on whether we are willing to force existing implementations to break ABI and increase object size for their existing client base.

Rank	i	o	Principle ID	E	G	H
12	1	@	MinExeSize	3	3	4
13	1	@	AllowOnAsyncFrame	-	-	@
14	1	@	ArgRestrict	-	-	@
15	1	@	MaxImpFlexAddNoex	9	-	9
16	1	5	MaxExpressInCode	1	1	1
17	1	5	MaxNegUnitTest	9	9	9
18	1	-	MinCostToFixStd	6	5	5
19	1	-	MinPermDiverge	9	9	@
20	-	@	MinObjSize			

We see little differentiation between E and G in the final 13 entries, whereas the primary reason for favoring H can be found in the ability to use `noexcept` to block throwing callback functions. Note that if we decide that we want the benefits of H and G combined, we can get add POR to H and call that policy statement I.

4.9 Recommendations

After looking at the first seven principles, we see that, of the original four basic strategies, a straight minimal use of `noexcept` without the Lakos Rule is optimal unless we are able to convince implementers to break ABI compatibility. If we decide G is possible and also want to allow `noexcept` for its alternate use to restrict function pointer parameters in frameworks, then that becomes the recommendation.

5 Conclusion

Inconsistent design decisions and time lost churning the development of new features are risks inherent to large-scale collaborative projects. The work of WG21 is a prime example of such a long-term, large-scale, geographically distributed, collaborative software-design project. Moreover, this massive project includes a Standard Library intended for wide reuse, which necessitates repeated, consistent, and correct application of language features to each new library component that is developed.

LEWG, responsible for evolving the Standard Library, has undertaken the task of providing a repeatable process for vetting and memorializing library polices so that they are readily accessible and maintainable over time. Still missing is a repeatable process for consistently and correctly selecting, from a set of proposed policy statements, the one best satisfies the principles deemed most important by the members of LEWG and the greater WG21 community and thus adds maximal value.

In this paper, we advocate use of the principled-design methodology described in [P3004] but with proposed solutions replaced with proposed policy statements. Encouraging this principled approach for both individual proposals and LEWG-sponsored policy-tracking papers has many distinct benefits over an unstructured format.

— Gratuitous variation in rendering is eliminated, making a policy submission paper and, especially, a policy tracking paper easy to understand and use.
— This approach provides a standardized way to compare policy statements, thereby conserving Committee time.
— This approach requires that competing policy statements be represented with motivating principles, which can then be fairly used for comparing all policy statements.
— Expecting a proposal author to fairly represent peers' proposed policy statements and to list valid concerns for their own proposed policy statement is an effective way of encouraging proposal authors to understand others' points of view.
— The work of characterizing principles can be done completed outside of and audited during Committee time, thereby streamlining the process.
— Committee time will be focused on determining the relative importance of motivating principles, which can then be applied objectively to all policy statements fairly.
— The scoring of solutions against principles can be performed ahead of time, even without knowing the principles' ranking. Once the rank is established, the compliance table can easily be ordered in real time to achieve the final product.
— The principled-design approach provides positive feedback by way of identifying reusable principles.

In short, this principled-design approach to adjudicating an optional policy provides an effective means of resolving difficult policy decisions that might otherwise be left to a majority vote, with minimal time required of the Committee while in session.

Most importantly, however, is that this process — like no other we know — all but guarantees that the policy statement chosen is truly optimal based on what we, as the C++ Standards Committee, determine to be most important principles for the successful and effective use of the language.

Finally, this process produces a work product that makes clear *why* a policy was adopted and what would be required to challenge a current policy, thus avoiding unnecessary discussion and conserving scarce Committee time.

6 Appendix: Lightly Worked Examples Of Papers

We now proceed to apply the ideas presented here to writing as examples a variety of policy papers for standards-proposals — each anticipating several competing viable policies. In each case, we'll enumerate a collection of possible policies and then use our principled-design methodology to advocate for just one or two of them. Each example could be separated into its own paper and, with just a little more work, be a viable WG21 policy proposal.

7 Paper 1: Under What Circumstances Is `constexpr` Employed?

7.1 Introduction

The `constexpr` specifier declares that the value of the function or variable may possibly be evaluated at compile time. In this section, we try to formulate a policy guiding the application of `constexpr` to elements affected by Standards Proposals.

Note that while similar policies may govern `consteval` and/or `constinit`, we are limiting ourselves to `constexpr` in this pedagogical example.

7.1.1 Principled Design

We follow the process described by [P3005]

1. State the question that must be addressed by a policy.
2. Describe each policy to be considered.
3. Collect the principles underlying the design discussion.
4. Rank and score the principles according to importance and objectivity.
5. Score the proposed policies against the principles.
6. Review the results, look for new policies, and repeat from step 3 if necessary.

7.2 Policy Question

When is it appropriate for an element of the Standard to be mandated to be `constexpr`? Should every possible interface be `constexpr`, or should concerns about expected compile time or the additional complexity of `constexpr` implementation be taken into account?

Note that this proposal is strictly about setting a policy for the Standard's use of `constexpr` - questions about whether `constexpr` should be implicit or about Circle-like metaprogramming models are out of scope.

7.2.1 Illustrative Example

Let's say someone is writing a paper proposing an extension to `std::string`, adding an `is_palindrome(void)` method.

```
    bool is_palindrome(void) const;
```

Should this method be `constexpr`?

Note that `cbegin` and `crbegin` are both `constexpr`, so there isn't really a technical bar to making this method `constexpr` as well in `C++20` or later.

7.2.2 Business Justification

1. We're trying to save time on each proposal that touches items that may need to be `constexpr`.
2. We're trying to increase the application consistency of `constexpr` across proposals.

7.2.3 Measure of Success

A successful policy will result in a straightforward and hopefully uncontroversial rule for the application of `constexpr`.

7.3 Probative Questions

1. Is the benefit of `constexpr` worth extra complexity in implementing a new feature?
2. What's the implementation and compilation-time cost of making something `constexpr`?
3. Is there a scenario where making this thing `constexpr` improves the resulting program?

7.4 Proposed Policies

7.4.1 Always `constexpr` When Possible

When a good implementation for the Standard element under consideration has no non-`constexpr` dependencies, use `constexpr`.

Motivating Principles

1. C++ should be as efficient as possible at runtime, even at the cost of compile time or implementation complexity.

Concerns

1. This could result in unsustainable compilation delays.
2. The implementation complexities for some interfaces may not be surmountable technically or economically.
3. Once the Standard declares something as `constexpr`, it's an irreversible changer, possibly precluding future improvements which would require a non-`constexpr` implementation.
4. `constexpr` method implementations must be fully inline, limiting opportunities for decoupling.

7.4.2 Never `constexpr` Unless Needed for Status Quo

Unless the standard element under consideration is already `constexpr`, do not add `constexpr`.

Motivating principles

1. Maximize compiler throughput.

Concerns

1. This would limit options for improving the library with better `constexpr` support.

7.4.3 Limit Mandatory `constexpr` to Methods having O(n log(n)) or Better Performance

When an implementation for the Standard element under consideration has no non-`constexpr` dependencies, and that implementation is at least as efficient as `O(n log(n))`, use `constexpr`.

Motivating principles

1. Compilations must not fail due to `constexpr` complexity.

Concerns

1. Slower methods may be the main region of interest in shifting costs to compile time with `constexpr` rather than runtime where possible.
2. The exact upper bound desireable may not be `O(n log(n))`.

7.5 Curated, Refined, Characterized, and Ranked Principles

Table 14: Importance and Objectivity of Principles

Rank	i	o	Principle ID	Principle Statement
2	9	@	runtimeEfficient	Minimizes runtime as much as possible.
3	@	9	maxCompSpeed	Maximize compiler throughput.
1	@	@	mustNotFail	Compilations must not fail due to `constexpr` complexity.

7.6 *Compliance Table:* Policies Scored Against Ordered Principles

Table 15: Key to Policies

Col		Policy
A	No policy	***<the empty statement>***
B	Original Proposal	**Always `constexpr`**
C	Original Alternate	**Never `constexpr`**
D	Original Alternate	**Limit `constexpr` by complexity**

Table 16: Compliance Table

Rank	i	o	Principle ID	A	B	C	D
1	@	@	mustNotFail	5	5	@	@
2	9	@	runtimeEfficient	_	@	1	@
3	@	9	maxCompSpeed	_	1	@	@

7.7 Analysis of the Compliance Table

Row 1 — A B C D — mustNotFail(@,@)

We see that always using `constexpr` may violate this principle, weakening the standing of this solution.

Row 2 — _ B _ D — maxCompSpeed(@,9) Always using `constexpr` is ruled out by this principle.

Row 3 — a _ C D — runtimeEfficient(9,@)

Never using `constexpr` is ruled out by this principle.

7.8 Recommendations

The best of the policies under consideration here would be to either limit `constexpr` to a certain algorithmic complexity threshold *(which is `O(n log(n))` pending discussion with compiler implementers)* or continue with no policy.

Further discussion of this proposal may lead to alternative policies and/or more principles, perhaps leading to a single clear policy.

8 Paper 2: When Should Hidden Friends Be Employed?

8.1 Introduction

A common way to implement a friend function for a type is to place those function's implementation (as a possibly templated free function) into the nearest enclosing namespace. This makes the friend function's name visible to qualified and unqualified name lookup.

If, in turn, the friend function implemented inside the class declaration, this name is not added to the enclosing namespace, but still visible to the ADL.

8.1.1 Principled Design

We follow the process described by [P3005].

1. State the question that must be addressed by a policy.
2. Describe each policy to be considered.
3. Collect the principles underlying the design discussion.
4. Rank and score the principles according to importance and objectivity.
5. Score the proposed policies against the principles.
6. Review the results, look for new policies, and repeat from step 3 if necessary.

8.2 Policy Question

A *hidden friend* is a `friend` function (possibly templated) whose definition appears inline within a class definition and that is not declared anywhere else. This paper provides a general policy on when the *hidden friend* idiom should be used and explores motivating principles behind this guidance.

8.2.1 Example 1: Out-of-class friend implementation

```
namespace A {

class Point {
    int d_x;
    int d_y;
    ...
    // Declare operator==() as a friend;
    friend bool operator==(const Point&,
                           const Point&);
}

// Declare operator==() in the namespace A
bool operator==(const Point&, const Point&);
}  // close namespace A

// Define operator==()
bool A::operator==(const Point& rhs,
                   const Point& lhs)
{
    return rhs.d_x == rhs.d_x && rhs.d_y == lhs.d_y;
}
```

8.2.2 Example 2: Hidden friend implementation

```
namespace A {
class Point {
    // Declare and define the operator==() in
    // the class declaration.
    friend bool operator==(const Point& rhs,
                           const Point& lhs)
    {
        return rhs.d_x == rhs.d_x && rhs.d_y == lhs.d_y;
    }
}
}  // close namespace A
```

8.2.3 Business justification

1. Increase compilation speed by reducing the size of the namespace overload sets.
2. Simplify idiomatic implementation of the friend functions.
3. Make friend function implementation less prone to errors (especially in the presence of templates.)
4. Avoid code duplication (definition is a declaration.)

8.3 Proposed Policies

8.3.1 Proposed Policy: Use hidden friends idiom when ADL is all that is needed

Hidden friend idiom shall be always used if at least one parameter of the friend function is associated with the class declaring the friendship.

8.3.1.1 Motivating principles

1. Reduce compile time.
2. Reduce an amount of boilerplate code necessary to implement a friend function.
3. Endorse correct usage.

8.3.1.2 Concerns

1. Might not apply some implicit conversions (is it really a concern???)
2. Friend's definition is visible.

8.3.2 Alternative Policy: Use hidden friends idiom for a fixed set of functions

Hidden friend idiom shall be always used for the set of specific function overloads (comparison, input/output, and arithmetic operators, swap, hash). Implementation of other friends is not specified.

8.3.2.1 Motivating principles

1. Endorse idiomatic implementation of the given set of friend functions.
2. The implementation is verifiable (aka - client will see implementation in class declaration.)

8.3.2.2 Concerns

1. The proposed set of friends is not clear.
2. Inconsistency
3. Might not cover a set of overloads desired for a DSL (Domain Specific Language) implementation.

8.3.3 Alternative Policy: Do not use hidden friend idiom

Do not use hidden friend idiom

8.3.3.1 Motivating principles

1. Minimize the surprising outcomes from library interfaces

8.3.3.2 Concerns

1. Friends implemented out-of-class are visible for the unqualified and qualified name lookup and may be sometime found by applying implicit argument conversion (which might be not desired)

8.4 Curated, Refined, Characterized, and Ranked Principles

We've refined the original motivating principles, preserving the order in which they are described in "Proposed Policies" We've characterized each principle's importance and objectivity. To save space, we've simply ranked the principles in place. Note that the rank will correspond to the row number in the final compliance table in the next section.

Table 17: Importance and Objectivity of Principles

Rank	i	o	Principle ID	Principle Statement
3	9	@	ReduceCompileTime	Reduce compile time
5	5	@	ReduceCodeSize	Reduce the size of the implementation
1	9	5	CorrectUsage	Make it hard to use incorrectly
2	5	5	Consistency	The solution should be uniform (no variations)
4	5	5	LeastSurprize	Minimize surpises from library interfaces

8.5 *Compliance Table:* Policies Scored Against Ordered Principles

Table 18: Key to Policies

Col		Policy
A	No policy	*<the empty statement>*
B	Original Proposal	**Use hidden friends idiom when ADL is all that is needed**
C	Original Alternate	**Use hidden friends idiom for a fixed set of functions**
D	Original Alternate	**Do not use hidden friends idiom**

Table 19: Compliance Table

Rank	i	o	Principle ID	A	B	C	D
1	9	5	CorrectUsage	8	9	9	8
2	5	5	Consistency	5	7	5	@
3	9	@	ReduceCompileTime	5	9	9	1
4	5	5	LeastSurpize	5	5	4	7
5	5	@	ReduceCodeSize	5	5	5	3

8.6 Analysis of the Compliance Table

Row 1 — A B C D — CorrectUsage(9,5)

Implementing a friend function using hidden friend idiom minimizes the chances of the implementation errors, specifically in the presence of templates, by coupling the declaration and definition with the context of the surrounding class. Out-of-class implementation have slightly higher chances of making subtle errors.

Row 2 — a B C d — Consistency(9,5)

Policy B provides a slightly better implementation consistency - whether or not a friend function is a subject to a hidden friend implementation can be simply inferred from the friend function signature. Policy C assumes that there will be a fixed set of friends that needs to be memorize. Policy A does not give any particular guidance and might lead to highly inconsistent implementations.

Row 3 — a B c d — ReduceCompileTIme(9,5)

While all 4 policies provide functionally equivalent implementation of the friend functions, hidden friend idiom can significantly reduce the namespace overload set that compiler uses to find matching signature for commonly used operators.

Row 4 — - B c - — LeastSurprize(5,@)

Policy B have slightly sharper "line" separating cases when the hidden friends idiom should be applied (based purely on function signature). Policy D is given slightly higher rating in this row, but its "less surprising" behavior is dictated by existing name lookup rules for out-of-class friends.

Row 5 — - B c - — ReduceCodeSize(5,@)

In general, hidden friend implementation requires less boilerplate code (forward declarations) that the out-of-class implementation.

8.7 Recommendations

Based on the results from the Compliance table, we would recommend using hidden friends implementation idiom for all friend functions having at least one parameter associated with the class declaring the friendship (Policy B).

8.8 Appendix

The information below is optional and helps to understand the benefits of hidden friend idiom with examples.

8.8.1 Reduce compilation time

Application of the hidden friend implementation idiom may significantly reduce a namespace overload set for commonly used operators and functions (comparison and arithmetic, “swap”, “hash”) which may lead to a measurable reduction of the compile time for large libraries with potentially huge overload sets.

8.8.2 Reduce amount of implementation boilerplate

Hidden friend implementation is in general less verbose that the equivalent implementation of the friend outside of the class.

8.8.3 Endorse correct usage

Hidden friend implementation make it harder to make a subtle errors when implementing friends for the class templates:

Wrong friend

```
namespace A {

template<class T>
class Point {
    ...
    // This is not the right friend!
    // Frienship is granted to a non-template
    // function taking Point<T>&
    friend bool operator==(const Point& rhs,
                           const Point& lhs);
}

}  // close namespace A

// Definition of the different operator==()
template<class T>
inline
bool A::operator==(const Point<T>& rhs,
                   const Point<T>& lhs) {
  ...
}
```

Correct usage

```
namespace A {

// Forward declaration of Point for the operator==
// template declaration.
template<class T>
class Point;

// Forward declaration for the friend
// declaration in the class.
template<class T>
bool operator==(const Point<T>& rhs,
                const Point<T>& lhs);

template<class T>
class Point {
    ...
    friend bool operator== <T>(const Point& rhs,
                               const Point& lhs); // **
}

}  // close namespace A

template<class T>
inline
bool A::operator==(const Point<T>& rhs,
                   const Point<T>& lhs)
{
   ...
}

// ** Older MSVC might not like the definition outside th
// See hidden friend implementation
```

8.8.3.1 Hidden friend

```
template<class T>
class Point {
    ...
    // Declare and instantiate the friend at
    // the same time with the main template
    friend bool operator==(const Point& rhs,
                           const Point& lhs)
    {
      ...
    }
}
```

9 Paper 3: Under What Circumstances Is `[[nodiscard]]` Employed?

9.1 Introduction

TBD

9.1.1 Principled Design

We follow the process described by [P3005].

1. State the question that must be addressed by a policy.
2. Describe each policy to be considered.
3. Collect the principles underlying the design discussion.
4. Rank and score the principles according to importance and objectivity.
5. Score the proposed policies against the principles.
6. Review the results, look for new policies, and repeat from step 3 if necessary.

9.2 Policy Question

It is desirable, sometimes, for the compiler to issue warnings whenever the return value of a certain value is discarded without being used.

This paper discusses possibilities such as the compiler automatically inferring the `[[nodiscard]]` attribute on some functions, or policies a team may take on which functions to manually mark `[[nodiscard]]`.

Marking functions `[[nodiscard]]` can go too far. In the C language, around 1991, 'lint' would issue warnings any time *any* function returned a value that was discarded, and there was no compiler switch to silence these warnings. It turns out that the `printf` family of functions all return a value that is nearly always ignored, so the only way to shut the warnings up was to `(void)`-out every `printf` call in your program. This turned out to be a serious impediment to persuading coders to get their code clean-linting.

Some functions take modifiable variables by reference or pointer through the argument list, and in those cases, the coder may be interested only in the effect on those variables and want to ignore the return value. Even if there are no such arguments, a function may have side effects on global state or do I/O, in which case, again, the coder may not be interested in the return value.

If a function has no side effects through the argument list or on global state or I/O, then the return value is the only service provided by the function, and there is no point in calling the function if it is ignored. Such functions are called 'pure' functions, and they should always be declared [[nodiscard]].

In the C++ language, if the implementation of a function is not visible in the present translation unit, the compiler itself is unable to determine whether a function has any side effects or does any I/O, and is unable to automatically make the function `[[nodiscard]]`.

There are other functions, such as memory allocators, that return a resource through the return value that must be managed, that is, released or freed. To ignore such a return value is a "leak", and almost always an error. However, the compiler is unable to recognize such functions, since memory allocators do effect global state.

9.2.1 Syntax

There are 3 ways to use `[[nodiscard]]`:

— Declaring a function `[[nodiscard]]`.

Will flag any call to the function that ignores the return value.

— Declaring a constructor `[[nodiscard]]`.

Will flag any object constructed by that constructor that is neither a variable nor a temp bound to a reference.

— Declaring a ‘class’ or ‘enum’ `[[nodiscard]]`.

  Will flag any instance where the ‘class’ or ‘enum’ is returned by value from a function and ignored. Note that if the ‘class’ or ‘enum’ is returned by reference, that is not flagged.

9.2.2 Illustrative Example of Constructive Use of [[nodiscard]]

Failure to declare variable name of RAII guard, causes RAII guard to be a temporary and release the resource it is meant to guard at the end of the statement rather than at the end of the block:

Correct code:

```
struct S {
    mutable std::mutex d_mutex;
};

void f(const S& s)
{
    std::lock_guard guard(s.d_mutex);

    ... critical section ...
}

int main()
{
    ... spawn many threads calling `f` ...
}

No compiler warnings.

Program works reliably at runtime.
```

Error: Forget to name guard:

```
struct S {
    mutable std::mutex d_mutex;
};

void f(const S& s)
{
    // The `lock_guard` is a temp within the
    // first statement.

    std::lock_guard(s.d_mutex);

    ... critical section ...
}

int main()
{
    ... spawn many threads calling `f` ...
}

No compiler warnings.

Runtime result: intermittent failures as multiple
threads enter the critical section at the same
time.  Programmer cannot understand what is wrong,
since the mutex appears to be locked.
```

9.2.3 Same Error, With [[nodiscard]] Warning:

Here we see how declaring an RAII constructor 'nodiscard' results in a compiler warning that would have saved the programmer in the previous example.

```
class MyLockGuard {
    std::mutex *d_mutex_p;

    // `nodiscard` constructor

    [[nodiscard]] explicit MyLockGuard(std::mutex& mutexArg)
    : d_mutex_p(&mutexArg)
    {
        mutexArg.lock();
    }

    ~MyLockGuard()
    {
        d_mutex_p->unlock();
    }
...
};

struct S {
    mutable std::mutex d_mutex;
};

void f(const S& s)
{
    // The `lock_guard` is a temp within the
    // first statement.

    std::lock_guard(s.d_mutex);

    ... critical section ...
}

int main()
{
    ... spawn many threads calling `f` ...
}

Compiler Warnings:

<source>: In function 'void f(const S&)':
<source>:14:27: warning: ignoring return value of 'MyLockGuard::MyLockGuard(std::mutex&)', declared with
   14 |     MyLockGuard(s.d_mutex);
      |                           ^
<source>:5:19: note: declared here
    5 |     [[nodiscard]] MyLockGuard(std::mutex&) {}
      |                   ^~~~~~~~~~

Which draws the programmers attention to the problem.
```

9.2.4 Justification

Certain functions and constructors have no useful, salient attribute other than their return value, so calling such functions and constructors when the return value is being discarded is pointless and is probably a programmer error that should be flagged with a compiler warning issued.

One example of a common misunderstanding is that the word `empty` is both an adjective and a verb. In the STL, it's an adjective accessor taking no arguments and returning a `bool` with information about the state of a container, but some coders misunderstand, thinking it's a verb meaning to remove all the elements in the container (that manipulator is `clear`). If `empty` is made `[[nodiscard]]`, these misunderstandings will be cleared up (since the coder would have been expecting a return type of `void` and get compiler warnings).

9.2.5 Measure of Success

Bugs caught with minimal programmer time, minimal false warnings.

9.3 Proposed Policies

9.3.1 Make all functions that return a value [[nodiscard]].

— Motivating Principles

(CatchRealErrors) Would flag all instances of return values being ignored.

— Concerns

(MinimizeQuantity) This would result in a huge number of warnings that would have to be silenced or ignored.

(MinimalEffort) The coder would have to spend a lot of time marking functions calls '(void)' when they return values that are not wanted.

9.3.2 Compiler automatically makes pure functions defined in translation unit as [[nodiscard]]

Functions defined in the translation unit known to have no salient characteristics other then the return value would be labeled `[[nodiscard]]` by the compiler.

— Motivating Principles

(MinimalEffort) The coder would not have to spend any time marking pure functions [[nodiscard]] if their definition is public.

— Concerns

(CatchRealErrors) Of functions not local to the translation unit, only functions where the full implementation is visible to the translation unit – inline and template functions – would be covered. When a non-template inline function is changed to out-of-line, [[nodiscard]] warnings would disappear in all translation units other than the one where the function is defined.

(MinimizeQuantity) If for some reason the coder didn't want some pure function to be [[nodiscard]] it would be hard to turn off. But such case would be infrequent (or even non-existent).

9.3.3 Compiler makes all constructors [[nodiscard]].

— Motivating Principles

(MinimalEffort) The coder would not have to spend any time marking constructors [[nodiscard]].

(GoodCoverage) To create an object and not call any methods on it or have it in a variable or bind it to a reference is almost always a mistake.

— Concerns

(MinimizeQuantity) It is conceivable that someone may have a constructor whose side effects are the point and the object created is not important – but that would be exceedingly rare, it's not the way C++ was intended to be written, and it would be more readable to just put that code in a free function. In these exceedingly rare cases (if any exist) the programmer can `(void)`-out the constructor call.

9.3.4 Library writers annotate all constructors of RAII guards & proctors

— Motivating Principles

(CatchRealErrors) Would catch a lot of errors.

(NotConfusing) Warnings would be easy to understand.

— Concerns

In most cases of calls to RAII guards and proctors, if the coder forgets to mention a variable name, the statement often deteriorates into syntactically incorrect most-vexing-parses that the compiler thinks are

definitions of uninitialized references, which result in confusing compiler error messages rather than the coder getting an easy to understand `nodiscard` warning.

(MinimalEffort) This would take some effort by library creators.

9.3.5 Compiler issues a warning if a pure function or constructor is not `[[nodiscard]]`

Description: If function is a constructor, or if it's pure (that is, has no salient effects other that the return value) and is not declared `[[nodiscard]]`, have the compiler issue a warning recommending that the function should be declared `[[nodiscard]]`. This would be controlled with a compiler switch, and the anticipation is that most builds would be with this switch disabled.

One possibility is that someone who does not have access to a compiler but who is writing an imperfect coding standards checker can implement something like this which will issue warnings just based on the interface of the function, and then the programmer can hand-inspect the function for side effects on global state.

— Motivating Principles

(MinimalEffort) This will help coders identity when they should add [[nodiscard]] to functions, but it will not identify all functions for which [[nodiscard]] would be appropriate.

— Concerns

(MinimizeQuantity) If this compiler switch is enabled when compiling legacy code, warnings will be very numerous. If, for some reason (hard to imagine) the coder does not want to make a pure function or constructor [[nodiscard]], there will be no easy way, other than pragmas (awkward), to silence the warning.

9.4 Combination Solutions

Solution 10.3.5 is incompatible with 10.3.2 and 10.3.3, since if the compiler is automatically making those functions '[[nodiscard]]", there is no need for them to be manually marked as such.

Similarly, 10.3.4 is incompatible with 10.3.3 since it would make no sense for library writers to manualy mark constructors '[[nodiscard]]" when the compiler has already done so.

9.4.1 Combined 10.3.4 and 10.3.5, library writers mark constructors of all

RAII guard objects `nodiscard` and compiler warns of pure functions not marked `nodiscard`.

— Motivating Principles

(CatchRealErrors) Would catch a lot of errors.

(NotConfusing) Warnings would be easy to understand.

(MinimalEffort) This will help coders identity when they should add [[nodiscard]] to functions, but it will not identify all functions for which [[nodiscard]] would be appropriate.

— Concerns

(MinimalEffort) This would take some effort by library creators. (MinimizeQuantity) If this compiler switch is enabled when compiling legacy code, warnings will be very numerous. If, for some reason (hard to imagine) the coder does not want to make a pure function or constructor [[nodiscard]], there will be no easy way, other than pragmas (awkward), to silence the warning.

9.4.2 Combined 10.3.2 and 10.3.3, have the compiler mark all pure functions and constructors `nodiscard`.

— Motivating Principles

(MinimalEffort) The coder would not have to spend any time marking pure functions [[nodiscard]] if their definition is public. The coder would not have to spend any time marking constructors [[nodiscard]].

(GoodCoverage) To create an object and not call any methods on it or have it in a variable or bind it to a reference is almost always a mistake.

(GoodCoverage) To create an object and not call any methods on it or have it in a variable or bind it to a reference is almost always a mistake.

— Concerns

(CatchRealErrors) Of pure functions not local to the translation unit, only functions where the full implementation is visible to the translation unit — inline and template functions — would be covered. When a non-template inline function is changed to out-of-line, [[nodiscard]] warnings would disappear in all translation units other than the one where the function is defined.

(MinimizeQuantity) If for some reason the coder didn't want some pure function to be [[nodiscard]] it would be hard to turn off. But such case would be infrequent (or even non-existent). It is conceivable that someone may have a constructor whose side effects are the point and the object created is not important — but that would be exceedingly rare, it's not the way C++ was intended to be written, and it would be more readable to just put that code in a free function. In these exceedingly rare cases (if any exist) the programmer can `(void)`-out the constructor call.

(MinimalEffort) The coder would not have to mark constructors as '[[nodiscard", and would only have to mark pure functions whose definition is not visible to all callers.

— Concerns

(MinimizeQuantity) It is conceivable that someone may have a constructor with side effects whose goal is other than the creation of the object, and the coder would have to '(void)'-out calls to such constructors

where a variable is not created. This would be **extremely** rare, since this is not the way C++ is intended to be written, and there would be no point in not making such functions free functions rather than constructors.

9.5 Curated, Refined, Characterized, and Ranked Principles

Table 22: Importance and Objectivity of Principles

Rank	i	o	Principle ID	Principle Statement
1	@	@	CatchRealErrors	Catch genuine programming bugs
2	@	@	MinimizeQuantity	Minimize quantity of new spurious warnings
3	@	5	NotConfusing	New compiler warnings are not confusing
4	5	5	MinimalEffort	Minimal programmer effort to mark [[nodiscard]]

9.6 *Compliance Table:* Policies Scored Against Ordered Principles

Table 23: Key to Policies

Col	Policy	Section
A	NoPolixy	
B	AllWithValue	(10.3.1)
C	AllPure	(10.3.2)
D	AllCtors	(10.3.3)
E	RAIIByHand	(10.3.4)
F	WarnPureAndCtorNot	(10.3.5)
G	`E` & `F`	(10.4.1)
H	`C` & `D`	(10.4.2)

Table 24: Compliance Table

Rank	i	o	Principle ID	A	B	C	D	E	F	G	H
1	@	@	CatchRealErrors	3	7	5	5	5	5	7	7
2	@	@	MinimizeQuantity	@	_	7	7	7	5	5	7
3	@	5	NotConfusing	@	@	@	@	@	@	@	@
4	5	5	MinimalEffort	3	1	7	7	3	3	3	7

9.7 Analysis of the Compliance Table

— A B C D E F G H

CatchRealErrors(@,@): Solution 'A' does particularly poorly.

— a B C D E F G H

MinimizeQuantity(@,@): Solution B fails this principle.

— a _ C D E F G H

NotConfusing(@,5): Everything passes this principle, so we'll ignore it.

— a _ C D E F G H

MinimalEffort(5,5): Solutions A, E, F, and G do poorly on this principle.

— a _ C D e f g H

— final analysis

Solutions C, D and H survive the culling as the best solutions. Solution H, being a combination of C & D, will catch more errors and is therefore superior to either of them.

9.8 Recommendations

The best policy would be for the compilers to make all pure functions and constructors [[nodiscard]].

10 Acknowledgements

Thanks to Brian Bi for his review ad for scoring the principles in our example.

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

11 References

[N2855] D. Gregor, D. Abrahams. 2009-03-23. Rvalue References and Exception Safety.
https://wg21.link/n2855

[N3279] A. Meredith, J. Lakos. 2011-03-25. Conservative use of noexcept in the Library.
https://wg21.link/n3279

[P0884R0] Nicolai Josuttis. 2018-02-10. Extending the noexcept Policy.
https://wg21.link/p0884r0

[P1656R2] Agustín Bergé. 2020-02-14. "Throws: Nothing" should be noexcept.
https://wg21.link/p1656r2

[P2267R0] Inbal Levi, Ben Craig, Fabio Fracassi. 2023-10-15. Library Evolution Policies.
https://wg21.link/p2267r0

[P2979R0] Alisdair Meredith, Harold Bott, John Lakos. 2023-10-13. The Need for Design Policies in WG21.
https://wg21.link/p2979r0

[P3004] John Lakos, Harold Bott, Bill Chapman, Mungo Gill, Mike Giroux, Alisdair Meredith, Oleg Subbotin. Principled Design for WG21.
https://wg21.link/p3004r0

[P3004R0] John Lakos, Harold Bott, Bill Chapman, Mungo Gill, Mike Giroux, Alisdair Meredith, Oleg Subbotin. Principled Design for WG21.
https://wg21.link/p3004r0

[P3005] John Lakos, Harold Bott, Bill Chapman, Mungo Gill, Mike Giroux, Alisdair Meredith, Oleg Subbotin. Memorializing Principled-Design Policies for WG21.
https://wg21.link/p3005r0

[P3085R0] Ben Craig. noexcept policy for SD-9 (throws nothing).
https://wg21.link/p3085r0