Principled Design for WG21

Document #:	D3004R0
Date:	2024-02-13
Project:	Programming Language C++
Audience:	EWG, LEWG

Reply-to: John Lakos
<jlakos@bloomberg.net>
Harold Bott
<hbott1@bloomberg.net>
Mungo Gill
<mgill83@bloomberg.net>
Lori Hughes
<lori@lorihughes.com>
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 presents a formal, systematic, individual- and group-friendly design selection and/or refinement methodology called *principled design*. We offer several examples of using this principled-design process to evaluate proposed solutions using relevant, characterized, and ordered principles that represent design criteria and to compare each criterion to each solution. By scoring each solution against each principle in turn, we can quickly elucidate which solutions are nonviable, ultimately guiding us to consensus and an optimal resolution.

2 Revision History

PREPRINT — THIS DOCUMENT WILL BE FINALIZED FOR THE FEBRUARY 2024 MAILING

R0 February 2024 (pre-Tokyo mailing)

This is the original version.

3 Introduction

With respect to software languages, *design-by-committee* might seem nearly impossible. Though each committee member is creating what they perceive to be the optimal, practicable design to address a given need or problem, each member's perception will, of course, derive from their own knowledge and experience. Thus, each member will naturally consider the principles relevant to their own work when developing design criteria. These principles will, in turn, be prioritized differently based on individually held and often unconscious biases, thereby leading to disagreement over what might otherwise eventually be accepted unanimously as a provably optimal solution.

The classical proposal-to-adoption approach involves discussion of a proposed change (e.g., to the Standard), followed by one or more five-way polls regarding level of "in favor" or "against" (SF, F, N, A, SA). This method typically leads to an optimal or at least acceptable decision in simple cases but often falls short when the problem involves a variety of relevant competing design criteria that, depending on prioritization, might lead to different conclusions. Some design principles are so critically essential as to be considered *imperative*; failing to adhere fully to even a single such principle would render a nonconforming proposed solution *nonviable* (with a consensus against). History shows, however, that the Committee is not infallible, and such imperative principles were sometimes given inadequate consideration or weight. In addition, because no such process is used nor the related discussion captured, someone who was not part of the meeting may have difficulty understanding how decisions were reached.

The principled-design process is remarkably intuitive. We introduce the basic idea via a lighthearted example that illustrates use of the process applied to a familiar nontechnical problem: selecting a venue for a company event.

Next, we provide detailed, step-by-step instructions for employing these techniques in practice. Early applications can be found in some SG21 papers, such as [P2834R1] "Semantic Stability Across Contract-Checking Build Modes" and [P2932R0] "A Principled Approach to Open Design Questions for Contracts."

In our second example, we illustrate how our principled-design methodology can be formally applied to a real-world problem, the seeds of which were already present in [P2723R1] "Zero-initialize objects of automatic storage duration," [**P2795R4?**] "Erroneous behaviour for uninitialized reads," and [P2754R0] "Deconstructing the Avoidance of Uninitialized Reads of Auto Variables." Together, our two introductory examples illustrate how the process is naturally transparent, informative, and auditable and how it may be followed as loosely or rigorously as need dictates.

Next, we discuss how a Standards meeting might be conducted to address proposals — especially those that have motivating requirements or criteria and/or some sort of principled-design analysis incorporated within them — for changes to the C++ (language or library) Standard. We expand on characterizing and applying principles to allow for differences of opinion within a C++ Standards subgroup. We also provide useful tips for how best to collect and process aggregate data in real time.

Finally, we provide in the appendix several examples related to real-world problems. For these examples, we collaborated with members of Bloomberg Engineering who are unfamiliar with WG21 methodology. These examples make our proposed principled-design methodology concrete and thereby facilitate its wide use within WG21 and beyond.

4 Example 1: Choosing a Venue for an Office Event

As a lighthearted introductory example, imagine a team of eight people working at a small company and tasked with recommending the venue for the company's end-of-year employee outing.

4.1 Capturing the Problem and Its Justification and Solution Metrics

Vik, the team leader, writes the task on the whiteboard.

Problem Statement: Select a venue for the end-of-year employee event.

Vik notes the justification for the event and how the success of the event will be measured.

Business Justification: Maintain or increase employee morale by fostering good working relationships and rewarding employees.

Measures of Success: Improved attendance, fewer complaints, and increased positive comments compared to the previous year's event.

4.2 Gathering Solutions, Motivating Reasons, and Potential Concerns

Vik then asks members of the team to think about potential venues for the outing and to email a suggestion, a reason motivating that choice, and any related concerns.

Vik soon receives several suggestions via email and gathers them into a document. Vik is personally hoping to visit the fancy steak restaurant that always gets great reviews.

— **Repeat last year's event**
Motivating reason: We have other work to do.
Concerns: Fewer people might attend, and the people who complained about last year's event would be ignored.

— **Dinner at a local fancy steak restaurant**
Motivating reason: A restaurant is a good place to talk, interact, and network.
Concerns: Will vegetarians have options?

— **Broadway play**
Motivating reason: Treat employees to a special event.
Concerns: Travel time to and from NYC will mean a late night.

— **Local botanical garden**
Motivating reason: Inexpensive, which saves money for the company.
Concerns: It might not feel like much of a treat.

— **Traveling roller-coaster park**
Motivating reason: This event comes to town only once every two years.
Concerns: Not everyone enjoys roller coasters.

— **First-run movie in local cinema**
Motivating reason: Relaxing and requires minimal effort from participants.
Concerns: Some might not consider it a special event.

4.3 Transforming *Motivating Reasons* Into *Principles*

Vik is pleased with the variety of suggestions. Vik wants to be fair and realizes that the motivating reason provided with each venue suggestion could also serve as a useful measure for evaluating all the suggestions. Vik copies each of these motivating reasons into the document, keeping them in the same order, and begins to refine them. For easy reference, Vik assigns each reason a short identifier.

— **Busy** — We have other work to do.
— **Talk** — A restaurant is a good place to talk, interact, and network.

— **Treat** — Treat employees to a special event.
— **Cost** — Inexpensive, which saves money for the company.
— **Timing** — This event comes to town only once every two years!
— **Easy** — Relaxing and requires minimal effort from participants.

Vik looks over these motivating reasons and thinks about how to make them into more generally applicable principles — either as predicates (inviting a binary, `true`-or-`false` answer) or metrics (that require a rating on some scale, say, from 0.0 to 1.0).

— **Busy** — Require no upfront design effort.
— **Talk** — Maximize interpersonal interaction.
— **Treat** — Maximize mean satisfaction of employees.
— **Cost** — Minimize monetary expense.
— **Timing** — Exploit limited-opportunity venues.
— **Easy** — Minimize effort of participants.

4.3.1 Refining and Augmenting the Sequence of Principles

Vik takes the list of venues and refined principles to the managers meeting, where three more constraints become clear.

1. The budget is set at $125.00 per person, is nonnegotiable, and must be used or forfeited, so selecting a venue that minimizes cost has no advantage.
2. By law, the venue must accommodate every employee regardless of disability or dietary restriction.
3. The company owner strongly prefers that the event end before midnight.

Vik makes the appropriate adjustment to the **Cost** principle, changing it from a *metric to be optimized* to a *binary predicate to be satisfied*. Vik then appends the two new principles **Access** and **Span** (each worded as a *predicate*, not a *metric*).

— **Busy** — Require no upfront design effort.
— **Talk** — Maximize interpersonal interaction.
— **Treat** — Maximize mean satisfaction of employees.
— **Cost** — Do not exceed $125.00 per person.
— **Timing** — Exploit limited-opportunity venues.
— **Easy** — Minimize effort of participants.
— **Access** — Accommodate every employee's needs.
— **Span** — End before midnight.

Vik reasons that not all of the principles are of equal importance, and some are not entirely objective. To coarsely quantify the *objectivity* of each principle, Vik decides to partition objectivity into three enumerated equivalence classes:

```
enum Objectivity { PROVABLY = 0xa, LARGELY = 5, NOT_REALLY = 0 };
```

Next Vik attempts to quantify each principle's *importance*. To faclitate discussion and agreement, Vik minimizes granularity but quickly realizes that to rank importance fairly requires additional enumerators. Vik settles on five graduated levels for characterizing the importance of each principle:

```
enum Importance { IMPERATIVE = 0xa, HIGH = 9, MEDIUM = 5, LOW = 1, IRRELEVANT = 0 };
```

4.3.2 Characterizing Principles: Importance and Objectivity

Vik then thoughtfully assigns both importance and objectivity values to each principle using the notation principle(importance, objectivity).

— **Lazy(IRRELEVANT, PROVABLY)** — Require no upfront design effort.
— **Talk(MEDIUM, LARGELY)** — Maximize interpersonal interaction.
— **Treat(HIGH, NOT_REALLY)** — Maximize mean satisfaction of employees.

— **Cost(IMPERATIVE, PROVABLY)** — Do not exceed $125.00 per person.
— **Timing(LOW, LARGELY)** — Exploit limited-opportunity venues.
— **Easy(MEDIUM, LARGELY)** — Minimize effort of participants.
— **Access(IMPERATIVE, LARGELY)** — Accommodate every employee's needs.
— **Span(HIGH, PROVABLY)** — End before midnight.

4.3.3 Auditing Principles and Their Characterizations

Vik and the team reconvene to discuss (audit) before attempting to apply each of these principles to all the solutions. Vik's team makes some good points.

— The **Busy** principle, like its accompanying solution, seems unhelpful to Vik personally, but his team insists **Busy** can improved by changing it from an *irrelevant* predicate to a *low-importance* metric.
— The fancy steak restaurant does have vegetarian dishes, so that concern is addressed, but the cost is typically over $200.00 per person, so the steak restaurant cannot be chosen.
— The team recommends instead an elegant Italian restaurant that's not quite as luxurious as the steak restaurant but is well within $125.00 per person budget and, in all other ways, is just as good.

Vik is happy to append the new solution — the Italian restaurant that derived from the original (steak) restaurant proposal — with its motivation being that the cost is not prohibitive.

Following the team's recommendation, Vik agrees to amend the **Busy** principle to be worded as a *metric* (as opposed to a *predicate*) rather than discard it entirely.

— **Busy(LOW, LARGELY)** — Minimize upfront design effort.

Vik then decides that the use of enumerators is overrated and tedious. Vik reverts to the literal numeric values but still adheres to the coarseness of the measurement scheme as a way to minimize the debate that would result from overly precise graduation.

— **Busy(1, 5)** — Minimize upfront design effort.
— **Talk(9, 5)** — Maximize interpersonal interaction.
— **Treat(9, 0)** — Maximize mean satisfaction of employees.
— **Cost(0xa, 0xa)** — Do not exceed $125.00 per person.
— **Timing(1, 5)** — Exploit limited-opportunity venues.
— **Easy(5, 5)** — Minimize effort of participants.
— **Access(0xa, 5)** — Accommodate every employee's needs.
— **Span(9, 0xa)** — End before midnight.

4.3.4 Ranking Principles

At this point, Vik ranks the principles based on (1) importance, (2) objectivity, and (3) a subjective *pair-wise comparison.*

Rank	Im	Ob	Principle ID	Principle Statement
7/**8**	1	5	Busy	Minimize upfront design effort.
4	9	5	Talk	Maximize interpersonal interaction.
5	5	0	Treat	Maximize mean satisfaction of employees.
1	0xa	0xa	Cost	Do not exceed $125.00 per person.
8/**7**	1	5	Timing	Exploit limited-opportunity venues.
6	5	5	Easy	Minimize effort of participants.
2	0xa	5	Access	Accommodate every employee's needs.
3	9	0xa	Span	End before midnight.

Vik observes that the rank was completely determined by just importance and objectivity except for **Busy(1, 5)** and **Timing(1, 5)**; Vik subjectively chooses **Busy** to be last, i.e., the rank of **Busy** is 8th.

4.4 Creating the *Compliance Table*

With the principles now refined, categorized, and ranked, Vik works alone (the team will again meet later to reach consensus) and applies, in ranked order, each principle to each solution. Vik realizes that more granularity will be needed to score solutions against the ranked principles and adds, for the compliance measurement only, two interstitial values — `MEDIUM_HIGH = 7` and `MEDIUM_LOW = 3` — beyond those for representing importance.

```
enum Compliance { TRUE        = 0xa,
                  HIGH        =   9,
                  MEDIUM_HIGH =   7,
                  MEDIUM      =   5,
                  MEDIUM_LOW  =   3,
                  LOW         =   1,
                  FALSE       =   0
                };
```

Vik creates a table — the *compliance table* — that will hold the final scores of each principle applied to every solution. The principles are in *rank order* in rows. Each solution is a column. Horizontal space is limited, so Vik assigns each solution an uppercase-letter key and reduces importance and objectivity to one lowercase character so they are distinct from the uppercase solution keys.

Given the complaints about last year's event, Vik makes the executive decision that having no event at all would be less offensive than ignoring the complaints and repeating last year's event. Vik changes solution A.

A. Status-Quo Default Solution — **Do nothing.**

B. Original Proposed Solution — **Dinner at a local fancy steak restaurant**

C. Original Alternate Solution — **Broadway play**

D. Original Alternate Solution — **Local botanical garden**

E. Original Alternate Solution — **Traveling roller-coaster park**

F. Original Alternate Solution — **First-run movie in local cinema**

G. Newly Proposed Solution — **Elegant Italian restaurant**

Rank	i	o	Principle ID	A	B	C	D	E	F	G
1	a	a	Cost	a	0	a	a	a	a	a
2	a	5	Access	a	a	a	a	3	a	a
3	9	a	Span	a	a	0	a	9	a	a
4	9	5	Talk	0	9	1	9	3	1	9
5	9	0	Treat	0	7	9	1	5	3	5
6	5	5	Easy	9	9	3	5	5	9	9
7	1	5	Timing	0	0	3	0	9	3	0
8	1	5	Lazy	a	5	1	7	5	7	5

Vik decides to represent the decimal value 10 (`0xa`) with just a single character, a, to minimize column width and to be distinct from A, the first and status-quo solution.

4.5 Auditing the Compliance Table

Vik then meets with the team to review the compliance table, and the team agrees that Vik's scores are at least plausible; Vik's score and each team member's opinion need not be in perfect agreement. Had significant disagreements occurred, then discussion would ensue.

4.6 Analysis of the Compliance Table

The team sees that the compliance table contains much valuable information. They review it row by row, starting at row 1 (i.e., the top-ranked principle). As they view each successive row, they exclude any solution that scores poorly unless it scores well in a streak of immediately successive rows, depending on the relative weight of the offending row.

Row 1 — A B C D E F G — Cost(a, a)

With the exception of solution B (the steak restaurant), all the solutions were within budget, each receiving a score of **a** (TRUE). B, however, was significantly over budget and therefore scored 0 (FALSE). Because this principle was scored as having **a** (imperative) importance and **a** (provably) objectivity, choosing A (doing nothing), the status-quo solution, would be preferable to B; hence, B can be excluded from further consideration. Note that this first principle offers no discriminating properties with respect to the other solutions.

Row 2 — A _ C D E F G — Access(a, 5)

All solutions except E (roller-coaster park) scored **a** (TRUE) against the second principle; E scored 3 (MEDIUM-LOW). Unless new information changes that score, solution E is no longer a contender. Note that, although this (binary) predicate is of **a** (imperative) importance, its objectivity scores only 5 (largely objective). Thus, providing a compliance value for a (binary) predicate, even when that principle is not fully satisfied, is useful for indicating relative partial credit, such as cases in which the principled-design process itself fails to deliver a single, clear winner.

Row 3 — A _ C D _ F G — Span(9, a)

Given all the remaining viable solutions have scored **a** (TRUE), the time limitations imposed by this high-importance and provably objective principle effectively serve to exclude C (Broadway play), having a score of 0 (FALSE), from further consideration.

Row 4 — A _ _ D _ F G — Talk(9, 5)

A key reason for holding such events is team networking, as measured by this high-importance design principle. Solution D (botanical garden) and our new solution, G (Italian restaurant), each scored 9 (HIGH). This high-importance principle, however, clearly eliminates for consideration A (do nothing) or F (movie), which received compliance scores of 0 (FALSE) and 1 (LOW), respectively.

Row 5 — _ _ _ D _ _ G — Treat(9, 0)

This final high-importance (9) principle serves as a strong discriminator between the two remaining viable options: D (botanical garden) and G (Italian restaurant). The botanical garden scored 1 (LOW), so the team would not consider it to be as much of a reward as the Italian restaurant, which scored 5 (MEDIUM). The team agrees that choosing the new solution, G, is likely but continues with the exercise.

Row 6 — _ _ _ D _ _ G — Easy(5, 5)

Solution G scores 9 (HIGH), significantly higher than the 5 (MEDIUM) score for the runner-up solution, D (botanical garden) for this medium-high principle.

Row 7 — _ _ _ D _ _ G — Timing(1, 5)

Both G (Italian restaurant) and D (botanical garden) score 0 (FALSE), and this principle scores 1 (LOW) in importance, so the team doesn't consider reviving any solutions it has already excluded.

Row 8 — _ _ _ D _ _ G — Timing(1, 5)

Finally, this principle admits a wide variety of scores, but its importance is 1 (LOW) and its rank is last, so the results are inconsequential. D (botanical garden) is slightly better than G (Italian restaurant) by a compliance score of 7 (MEDIUM-HIGH) to 5 (MEDIUM).

4.7 Choosing an Optimal Solution

The team compares solutions D (botanical garden) and G (Italian restaurant) one last time, just to be sure.

Rank	i	o	Principle ID	D	G	Notes
1	a	a	Cost	a	a	
2	a	5	Access	a	a	
3	9	a	Span	a	a	
4	9	5	Talk	9	9	G == D
5	9	0	Treat	1	5	G > D
6	5	5	Easy	5	9	G > D
7	1	5	Timing	0	0	G == D
8	1	5	Lazy	7	5	G < D

Looking again at the table, one team member argues that, for the Talk principle, an indoor venue might even be slightly more conducive to talking than an outdoor one. The team concludes that G, the newly proposed solution of the elegant Italian restaurant, is clearly the optimal choice.

5 Writing a Principled-Design-Based Standards Proposal

Let's follow out simplified example with step-by-step instructions for crafting a Standards proposal in accordance with the principled-design methodology. Following the principled-design process will allow Standards Committee members to rapidly understand the proposed change and set of alternative solutions being considered, the respective motivations for considering them, the concerns associated with each solution, and how well each solution satisfies all the motivating (and other) principles (i.e., the criteria that pertain to the solutions and against which the solutions are measured.)

This section provides an outline of how to capture the principled-design process when determining if and to what extent a given solution is superior to other alternatives, one of which is always the status quo (i.e., the Committee's lack of consensus for change). Section heads that appear in angle brackets (< >) indicate sections that would be included in a principled-design proposal; those without angle brackets are simply part of *this* paper's description of the process. A real-world example of our principled-design methodology — applied to the elimination of undefined behavior (UB) that would result from reading uninitialized nonclass automatic variables — will follow in "Example 2."

5.1 <Brief Historical Recap>

After providing the basic sections, namely an abstract, revision history, and perhaps a brief introduction, the next step is to summarize what discussion and changes have occurred to date, i.e., how we got to today's state. From this point on, the paper follows a tightly specified framework.

N.B. — When writing the very first proposal that addresses a problem, authors will need to describe the issue in more detail (not so briefly) and might need to identify principles before developing potential solutions.

5.2 <Problem Statement>

In the first formal part of our principled-design-based proposal-writing process, the author states clearly what needs to change. The examples in this paper focus on reducing or eliminating the likelihood of executing undefined behavior in modern C++ programs, but any addition or modification of a C++ feature is fair game for this process. In fact, this approach to choosing a best solution, as we saw previously, is not limited to WG21, programming language design, or even technology. These principled arbitration techniques can be applied to any problem having multiple proposed solutions.

5.2.1 <Illustrative Example>

Any worthwhile proposal will elucidate the issue via a telling example.

1. The text preceding the example should tell readers what they are about to see.
2. The text immediately before the example should call out by name the item or point in the example to which the author wants to draw the readers' eye.
3. The example should include copious comments stating what is happening (often including the obvious). Readers, especially those who learn best by example, will appreciate useful annotations.
4. The text immediately following the example should reiterate what the reader just saw.

As a concrete example of how we would write a readable example, we have adapted some text and reprinted other text and a code snippet from our recently published book, *Embracing Modern C++ Safely*.[1]

> When a variable is declared within the body of a function, we say that the variable is declared at **function scope** (a.k.a. **local scope**). An object (e.g., `iLocal`) that is declared `static` within the body of a function (e.g., `f`) will be initialized the first time the **flow of control** passes through the **definition** of that object:

```
#include <cassert>  // standard C assert macro

int f(int i) // function returning the first argument with which it is called
{
    static int iLocal = i;  // Object is initialized only once, on the first call.
    return iLocal;          // The same iLocal value is returned on every call.
}

int main()
{
    int a = f(10);  assert(a == 10);  // Initialize and return iLocal.
    int b = f(20);  assert(b == 10);  // Return iLocal.
    int c = f(30);  assert(c == 10);  // Return iLocal.

    return 0;
}
```

> In the simple example above, the function, `f`, initializes its `static` object `iLocal` with its argument, `i`, only the first time it is called and then always returns the same value, e.g., 10. Hence, when that function is called repeatedly with distinct arguments to initialize the `a`, `b`, and `c` variables, all of them are initialized to the same value, 10, supplied to the first invocation of `f`. Although the function-scope `static` object, `iLocal`, was created after `main` was entered, it will not be destroyed until after `main` exits.

5.2.2 <Business Justification>

Every idea is potentially good, but not every good idea is superior to and can displace another good idea. The C++ Standards Committee has limited time to process proposals and needs to know in advance that the author's idea has the potential to provide a return on investment that's worth consuming valuable Committee time.

For example, suppose the Committee has time to consider only one of two proposals in a given meeting. Which of these should be chosen?

1. Attempt to eliminate the security vulnerability associated with uninitialized variables on the program stack.

2. Provide a more streamlined syntax for indicating the various (named) C++-style casts that are, say, more reminiscent of C-style casts.

In the first case, the author can cite the various safety and security reasons that affect individual customers and the health and longevity of the C++ language as well as the concerns that the issue might in some cases

[1]See "Function `static` '11," [lakos22], p. 68.

affect either runtime performance or the ability to statically detect inadvertent errors. In the second case, the motivating reasoning is to make the C++-style cast syntax expressible in fewer characters, and the concern is that the syntax would no longer be as easily searchable. Given these two proposals, the business justification for the former seems solid and for the latter is less clear.

In short, proposal authors must convince the Committee that their proposal has practical (e.g., economic, ergonomic, safety, correctness, or productivity) value.

5.2.3 <Measure of Success>

The proposal author might think the properties of their solution are obvious but should take the time to spell them out. Think of these properties as postconditions to the design process. What must hold if the solution is adopted?

For example, if we want to eliminate all or some of the undefined behavior resulting from a particular aspect of C++ programming without rendering the feature substantially less useful, let's state that explicitly.

A successful solution will

— eliminate or substantially reduce the undefined behavior associated with accidental misuse of this feature
— not prevent, inhibit, or substantially degrade effective use of the feature

These goals state what must be true to sufficiently address the stated business need and what is required of a successful solution. A proposed solution that fails to substantially accomplish any of these goals is nonviable even if it fully complies with all prevailing relevant principles.

5.3 <Probative Questions>

An important benefit of our principled-design process is its ability to determine how well a potential solution can be applied to real-world examples. Because exactly what principles might turn out to be relevant to the problem is often initially unclear and to avoid prematurely engaging in wordsmithing general principles, an early part of the process is introspection and discussion (e.g., on the reflector) about the kinds of questions that will inform the principle-writing step. A simple question about the solution, such as, "Does a solution that includes X imply Y?" is a great starting point.

We will accumulate a canned list of such questions over time. Here are just a few we often find useful.

— Does your solution affect
    — backward compatibility
    — defined behavior
    — what compiles
    — compile time
    — run time
— Does it require a new keyword?
— To what extent does your solution have implementation experience (e.g., at scale, which compilers)?

5.4 <Solutions>

5.4.1 <Proposed Solution: *Title of Solution*>

A Standards proposal using principled design is the result of a well-reasoned evaluation of all the known available options. A preferred choice arises from the proposer's evaluation, based on their perception of the relative importance of all the relevant principles applied to each solution considered — i.e., the status quo, proposed, alternate, and new solutions. This is where traditional advocacy would be placed. Showing bias here is expected since the author clearly represents that, of all the solutions considered, their proposed solution is superior to the status quo and to the alternative solutions and is unlikely to preclude some better solution in the foreseeable future. The author should describe the proposed solution is sufficient (but no more) detail to distinguish it from alternative solutions.

As an example, consider the `noexcept` specifier.[2] The purpose of this feature is to let users of (typically) a move constructor indicate that no exceptions will be thrown from the function. One of the subdecisions of this feature's design is what should happen if the `noexcept` specifier is added to a defaulted special member function.

```
struct X {
  X(X &&) noexcept = default;
    // This move constructor is declared to not throw.
};
```

Consider these options for the syntax.

1. Ill formed
2. Undefined behavior
3. Ill formed if the deduced value isn't the same as the explicit one
4. Well formed, but the function is deleted if not the same
5. The explicit value overrides the deduced value.

One might think the answer is obvious, and it would have been had principled design been applied. Sadly, principled design was not well known when the feature was conceived; C++11 employed option 3, C++14 employed option 4, and since C++17, we have used option 5.

<Motivating Principles>

The process of principled design now begins in earnest. Let's start by simply focusing on the idea of the principle, i.e., a criterion against which a solution can be measured.

Clearly, the proposal author has several reasons for preferring one solution over the others, and this section is the place to state that as a *measure*, not as an opinion. The author provides a principle — an criterion for which this solution will score perfectly (or at least highly).

The principle might be either a binary "yes" or "no" or a quality to optimize. Stating a principle as an absolute is allowed and, because our scoring scheme is graduated, the author can say something like "The solution is safe" and then gauge how well the solution satisfies that binary (albeit less-than-objective) principle. An important aspect of this process is to avoid repeating motivating principles. To that end, the author mentions all the motivating principles that apply to the preferred solution. Subsequent alternate solutions will then focus on only those principles that differ from the proposed solution.

The objectivity of the criteria is also considered. Saying that a solution is safe invites the question of what safety means. Generally speaking, the more objective the principle, the more useful for identifying a clearly best solution. (We'll discuss that in "Scoring Objectivity.")

Furthermore, not all principles are of equal importance. Some principles are imperative, some are of high importance, some are of only medium or low importance, and, of course, many might be irrelevant. (We'll present more on that topic in "Scoring Importance.")

Let's now consider a few examples of principles.

- **Binary Property vs. Quantitative Measure**
  - Binary: All code that previously compiled will, if it compiles under the new solution, behave as it did previously, i.e., as if this solution were not adopted.
  - Quantitative: Minimize the code that has different observable behavior.
- **Objectivity**
  - Provably Objective: All code that previously compiled will continue to compile.
  - Largely Objective: All code *that is likely to occur in practice* and that previously compiled will continue to compile.
  - Subjective: All *useful* code that previously compiled will continue to compile.

[2]See "Nonthrowing-Function Specifier," [lakos22], p. 1085.

- **Importance**
  - Imperative: All previous code with well-defined behavior, provided it continues to compile, is backward compatible.
  - Highly Important: All defect-free code that previously compiled continues to compile.
  - Moderately Important: All runtime defects of this ilk are caught at compile time.
  - Minimally Important: The solution doesn't favor [a particular group of] developers.
  - Irrelevant: The choice of syntax doesn't affect run time.

For now, the goal is to express what's good about this solution with as many relevant, positive, and motivating properties — especially important, objective, and/or binary ones — as the author deems worth considering. (We'll discuss perfecting the principles in "Refining the Principles.")

<Concerns>

We are all, of course, trying our best to arrive at a provably best solution. Proposal authors are expected to present valid concerns that a solution — even their proposed solution — might violate other, presumably less important principles. The content of this section should inspire colleagues to develop an improved solution that encompasses the best aspects current solutions *and* addresses the identified concerns, e.g., the Italian restaurant option in our first example. We'll model this again in our next example ("Example 2").

For example, reasonable concerns about a solution could be that it (1) doesn't have any known implementation experience, (2) might have an unacceptably high impact on runtime performance, or (3) deliberately trades off some important properties (e.g., preserving historical intent versus reducing the ability to ensure correctness at compile time) for others (e.g., simplifying the language versus minimizing accidental security vulnerabilities).

5.4.2 <Alternate Solution: *Title of Solution*>

Authors will repeat this section for each alternate solution. (The first solution is always the status quo, representing no consensus for change, and we'll discuss that more in "Default, Proposed, and Alternate Solutions".) The author will almost certainly have considered other alternative solutions, compared them against their preferred solution and, for whatever reason, decided that the alternate solution was inferior. Yet the author must have some reason to have considered the alternate solution, such as it (1) solved the problem sufficiently and perhaps (2) provides a benefit that was in some way arguably better than one or more aspects of the preferred solution.

The authors will ideally explain each solution in sufficient detail for the Committee members to understand what's being suggested. Having a suite of related solutions that differ in only a small aspect, however, is common. Each of these variants is worth mentioning, and if the variants affect the choice, the author(s) are encouraged to separate them into one or more preferred variant(s) from the others.

<Motivating Principles>

The goal is to provide any principles — beyond those listed with the preferred solution and regardless of perceived objectivity and relative importance as long as they are supportive, somewhat plausible, and relevant — that favor this alternative solution. If the author suspects that this alternate solution will score better on at least one of the principles stated above, then this solution is fair game. Even if this solution is simply popular and the author is convinced it will score poorly, the author is well advised to include it so it can be objectively scored and transparently compared to all the other potential solutions.

<Concerns>

Again, authors must scrupulously identify any issues or flaws associated with the alternate solutions — especially any concerns that might not have been addressed by the principles.

5.5 <Curated, Refined, Characterized, and Ranked Principles>

At this point, most of the research has been completed, and the author is ready to collect, refine, and score the motivating principles for each solution.

5.5.1 Collecting the Raw Principles

The authors collects all the motivating principles in the order in which they appear in the solutions sections, i.e., *chronological* order, and lists them verbatim.

5.5.2 Refining the Principles

The author then reads the principles to either ensure that they are optimally worded or refine them as needed. The author might want to (1) tighten the language, (2) separate principles as needed for specificity, (3) combine them as needed for generality, (4) remove those deemed irrelevant, or (5) append entirely new ones based on more careful consideration. The refined principles can and typically will deviate from their original versions in the paper. The goal is to make the refinement process part of the paper so Committee members can follow the development of the principles. That transparency is valuable and essential: It allows the *original* motivating principles to be local to the proposal and the *refined* principles to be applicable in general. Importantly, the author should not go back and replace the original motivating principles; authors should show their work.

1. Generalize and clarify the principle so that it applies to all solutions equally.

   For example, one might remove verbiage that applies to only the specific solution.

   — Before: The compile time of function `foo` is minimized.
   — After: Compile-time for this feature is minimized.

2. Remove vague language that reduces objectivity.

   For example, one might change "does not typically" to "does not."

   — Before: There is no change in useful runtime behavior.
   — After: There is no change in runtime behavior.

3. Prefer a binary property to a metric one where the former is more accurate.

   For example, one might change "Minimize X" to "Require that X < Y" (or perhaps keep both but at different priorities).

   — Before: Minimize the time span of the event.
   — After: The event will complete no later than midnight.

4. Prefer a metric property to a binary one when that property is not binary.

   For example, one might change "X does not affect Y" to "Minimize the effect of X on Y."

   — Before: The event will be pleasant.
   — After: Maximize the pleasantness of the event.

5. Divide principles that are insufficiently focused on a single aspect.

   For example, one might divide compile-time and runtime comparison into two principles.

   — Before
      — Maximize performance.
   — After
      — Minimize compile time.
      — Minimize link time.
      — Minimize run time.
      — Minimize development time.
   — Before
      — Minimize backward incompatibility.
   — After
      — Minimize incompatibility at compile time.
      — Minimize incompatibility at run time.

6. Combine principles that are essentially equivalent unless one is a special case of another and has substantially different importance. Keep the principles in *chronological* order, even after they are refined.

   For example, combine principles that rephrase one another.

   - Before
     - Optimize runtime performance.
     - Reduce run time.
   - After
     - Minimize run time.
   - Before
     - Easy to understand
     - Easy to teach
   - After
     - Minimal learning curve

   However, ensure the principles really are equivalent before combining them. Do not, for example, combine "backward compatible" with "mutually substitutable."

7. Double-check all the concerns and make sure no principle was omitted. Add unsaid but assumed principles.

   - Is implementable
   - Has associated implementation experience
   - Does not preclude other viable (arguably as good or better) solutions
   - Does not violate any of the implicit principles set forth in D&E

5.5.3 Characterizing the Principles

The author now has a *chronological* sequence of principles that is complete enough to demonstrate (or not) that the proposed solution is objectively better than the alternatives. The author next includes the principles in a table.

Importance	Objectivity	Principle ID	Principle Statement

Initially, the only field with content will be Principle Statement. The author then chooses a short descriptive identifier, the Principle ID. (This short name will be used later in the *compliance table* during the scoring of solutions.) For example, if the Principle Statement is "Minimize typical run times," the Principle ID might be "minRunTimes."

5.5.4 Limiting Resolution Per Category

The next step is to associate with each principle two characters that identify (1) its *importance* and (2) its *objectivity*, respectively. Scoring is not as easy as one might think. A tension between precision and consensus needs to be addressed, and not all field values make the same tradeoff. The idea is to put each answer on the real number range from 0 to 1, where 0 is identically `false`, 1 is identically `true`, and everything else is somewhere in between. That said, we have no justification for the level of precision in values like 0.385. In fact, in many cases just 0 or 1 is sufficient. Moreover, finer granularity introduces more opportunity for subjective disagreement.

The discrete answer ranges here have been determined impartially through actual use and, by all accounts, seem to work well in our practical experience so far. We have chosen a fixed palette of single-character answers that work for all three categories, i.e., *objectivity*, *importance*, and (later) *compliance*. The elements valid for each successive category are a proper subset of the next.

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
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

We use a single character for compactness, and we chose not to use a digit for values that represent True, False, or Unknowable. The choice of @ for True or 100% is to consume maximal real estate to stand out and derives from the hex digit **a** for decimal 10. The choice of the hyphen character (-), used to represent `false` or 0%, was chosen so as to also stand out (by consuming minimal real estate) as well as being centered and a common character.

A question mark symbol (?), used later for compliance-table entries only, means that the scorer believes that an accurate value is not easily knowable in the foreseeable future. Also note that symbols 2, 4, 6, and 8 — representing interstitial values for the compliance table only and indicated with n/a in the table above — are reserved for the rare case in which expressing the relative ordinal values across solutions for a given principle is essential, but they are otherwise to be avoided to prevent debate regarding questionable precision.

5.5.5 Scoring Objectivity

The purpose of the objectivity field is to determine if everyone can agree on whether a solution satisfies a principle. Simple binary questions of fact are typically objective. For example, if the principle were "Does not affect the defined behavior of any previously well-formed programs," the principle either would or would not be satisfied, no subjectivity is involved, and the principle scores @ for objectivity. Another example would be, "There is no change to defined behavior."

The wording of a principle can affect its objectivity, and the principle should be written to be as objective (and generally applicable) as possible. For example, "Has no effect on the observable behavior of real-world programs" is not entirely objective because what constitutes a real-world problem leaves room for interpretation. By contrast, "Minimizes the affect on changes to observable behavior" is worded differently, yet this principle is quite objective because, if a solution has literally zero effect on observable behavior, one could state objectively that the solution scored @ or 100%, where many other principles that start with "Minimize" or "Maximize" can get 9 at best or 1 at worst.

Writing every important principle provably objectively is not always possible, yet even somewhat subjective principles can be important and useful. For example, "Maximizes runtime performance," "Minimizes compile time," and "Minimizes object size" aren't absolutes and thus would never receive @ or - scores (unless the author could somehow prove that they had reached a best-possible result), but some solutions *can* score 9 and others 7, 5, 3, or even 1. These somewhat objective, metric scores are helpful information.

Finally, some principles are inherently so subjective that they are unlikely to be especially useful because the answers to the questions will vary widely with the opinion of the individual who is scoring a solution against that principle. For example, if the principle were "The increased ease of use justifies the increase in typical run times" (or vice versa), the answers would vary widely from one scorer to the next, making the results unreliable and hence less desirable as a means of demonstrating conclusively a best solution. Such a principle would, therefore, be scored as having the lowest possible objectivity rating (-), meaning low (but not necessarily zero) objectivity.

5.5.6 Scoring Importance

The importance category has five values. At the extremes are *imperative* (@) and *irrelevant* (-). If a principle has imperative importance, any solution that fails to satisfy the principle is rejected over the status quo (do nothing) unless the status quo solution itself is already somehow in violation of the same imperative principle. If a principle is of irrelevant importance, its objectivity score is moot because the principle will not contribute anything to the final decision of which solution is optimal.

All other principles are partitioned into *High* (9), *Medium* (5), and *Low* (1) importance levels. Further gradation of importance would be superfluous.

— (1) Imperative: All defined behavior that continues to compile remains unchanged.

— (9) High: All well-defined code that previously compiled continues to compile.

— (5) Medium: Runtime performance does not decrease, even when the new feature is used explicitly.

— (1) Low: The feature is not easily confused with similar (but not identical) features in other languages.

— (-) Irrelevant: Use of this feature mitigates risk in asset-backed securities.

The author uses objectivity to help break ties between principles with the same importance score. After that, the author can perform a linear pass over all the principles and swap the order of any that they feel were not properly ranked by just importance and objectivity.

Note that relevance of a common principle will typically depend on its particular application. For example, “Improves runtime performance,” sometimes by an order of magnitude, will apply to, say, PMR allocators, but will be irrelevant to the application of `noexcept` to functions that are never queried using the `noexcept` operator.

5.5.7 Ranking the Final Principles

Once the author has assigned both objectivity and importance scores to each of the entries in the refined-principles list, they can order the principles based first on importance, then objectivity, and finally a subjective pair-wise comparison. The highest objective rank (importance, objectivity) is (@, @), followed by (@, 5), (@, -), (9, @), and so on down to (1, @), (1, 5), (1, -), (-, @), (-, 5), and finally (-, -). This deliberately coarse ranking allows for both sufficient resolution and the possibility of partitioning into distinct equivalence classes with a reasonable hope of consensus. Ties will occur, and the author will then be determine the rank via a subjective pair-wise comparison within each equivalence class. In this way, the rank from 1 to N is determined.

Importantly, once the author has refined, removed, split, and added principles and then categorized and ranked them, they leave the principles in *chronological* order so readers can easily retrace the thought process from the initial to the refined principles.

Rank	i	o	Principle ID	Principle Statement

5.6 <The Compliance Table: Solutions Scored Against Ordered Principles>

Now each principle ID will be listed in the compliance table according to its rank, i.e., in a strictly decreasing order of priority. Before the author renders the table, however, they recreate each of the solutions, mapped to a single capital letter in a column proximately to the table for quick reference.

5.6.1 Default, Proposed, and Alternative Solutions

The author now collects the set of possible solutions for quick reference, mapping each in well-defined order to a sequence of capital letters, and continues until all solutions have been listed, listing last any new solutions that arose from the analysis itself.

Col.	Solutions
A	Status-Quo Default
B	Original Proposed
C	Original Alternate
...	...
N	Newly Proposed

5.6.2 The Compliance Table

The form of the compliance table combines the mapping described in the previous subsections.

Rank	i	o	Principle ID	A	B	C	D	E	F	G
1										
2										
3										
4										
...										
N										

The author is now ready to apply each of the principles, ordered by highest priority first, to each solution in turn. We offer some guidance about scoring both kinds of principles.

1. Typical compliance scores are @, 9, 7, 5, 3, 1, and - with 8, 6, 4, 2, and ? in reserve to be used only if necessary.
2. Binary (yes/no) questions are designed to be answered with either @ or -, but any value may be used when that answer is more descriptive.
3. Metric questions, however, can be trickier since the boundary values @ and - might be inappropriate. For example, the principle "Maximize implementation experience" could reasonably be scored in the range of [9 ... -], with - meaning literally zero implementation experience. The use of @, however, would be hyperbolic, implying infinite implementation experience.

As an illustrative example, let's revisit the C++11 changes to **function-local static-duration** object initialization, shown in "<Illustrative Example>," and how these changes would score against the highly important (9) principle of "Do not lessen runtime performance." For background information, see [N2660]. For this principle, the status quo would score @ since performance is unchanged. The [N2660] proposal would, in fact, lessen runtime performance, but the *slow path* would be followed at most only once per variable during initialization. The more common case would follow the *fast path*, so a score of @ is unjustified, but a score of either 7 or 9 is reasonable.

5.7 <Analysis of the Compliance Table>

Now the author can analyze the table, row by row, starting from the top. The rows are ordered from highest priority to lowest, so any solution that fails to satisfy an early row might be an appropriate candidate for rejection before considering additional rows. If a solution scores poorly (though not so badly as to warrant outright rejection) on a single row but very highly on multiple subsequent rows, then the author might choose to retain it in the candidate pool if it is clearly better than other candidates when considering subsequent principles. As the author analyzes each row, they must document in the paper which candidate solutions remain, which have been rejected, and which are maybes, stating clearly our thought processes and reasons behind the decision to reject or retain.

As the author works their way down the list, only one or maybe two solutions will remain as clearly viable candidates worthy of consideration. These relatively promising solutions should be discussed and contrasted in further detail.

5.8 <A Better Solution — Title of Solution>

After reviewing the compliance table, the author will often see that a promising solution was disqualified early due to failing just a single important principle. Combining the knowledge from another principle into a hybrid solution or creating a new derivative solution that removes the failure to comply to that important early principle can often result in a clear winner. In "Example 1," the originally proposed venue would have exceeded the budget but scored well on the other principles and warranted further consideration, leading to the alternative proposal of a similar but less expensive venue. As shown in "Example 1," the work invested to arrive at the new solution should be shown; i.e., don't rewrite the lab notebook.

<Motivating Principles>

When presenting a new competing solution principle, the author should make sure that all its additional strengths are captured by its motivating principles, but new refined principles are needed only if they are not already represented.

The original proposed solution and the alternative solutions are often designed without advance knowledge of the principles that would be applied. New solutions are designed with hindsight and a more complete understanding of the relative advantages and disadvantages of the previous alternatives.

<Concerns>

Again, any drawbacks to this new approach should be fully disclosed.

5.9 <Recommendations>

Lastly, the author should recommend what they conclude is the best course of action. Sometimes more than one solution is acceptable. A common situation is that the proposed solution seems likely, but the Committee lacks sufficient confidence to adopt it because it might preclude other potential solutions that could prove to be better in the future. In such cases, the recommendation should provide a sequence of solutions that are increasingly less aggressive and restrictive until they no longer preclude all reasonable final solutions or reach the status-quo solution. Waiting until more research is done or until the need becomes more acute is a possible outcome.

6 Example 2: Case Study — Use of Principled Design in WG21

Our systematic, principled-design approach might seem novel, but it's already in use in WG21. For example, SG21 has employed an evolving form of this discipline throughout 2023 to establish definitive design imperatives that govern consistent design across myriad features of the Contracts MVP, currently on track for C++26. Moreover, an early incarnation of this approach was used to good effect in EWG to evolve and eventually adopt a proposal establishing a new behavior category, *erroneous behavior*, that can be used to extinguish a wide range of critical security vulnerabilities endemic to C and C++ while avoiding debilitating side effects and even improving correctness through enhanced testability.

6.1 Brief Historical Recap

In November 2022, JF Bastien presented a proposal — [P2723R0], which he dubbed his "opening bid" — to plug a major security vulnerability stemming from frequent uninitialized-read errors of (automatic) variables on the program stack. His proposed solution — initializing nonclass variables to zero by default — all but eradicated that security vulnerability but with arguably acceptable potential added runtime overhead. On the other hand, Bastien's proposal had the unfortunate (and arguably fatal) drawback of permanently rendering the abject incorrectness of such defects no longer mechanically diagnosable, which would include existing defects in legacy code.

An entirely new idea (*erroneous behavior* or EB) — invented by Thomas Köppe in response to Bastien's challenge and first mentioned in fall 2022 on the WG21 reflector— enabled a demonstrable improvement over Bastien's original design. In February 2023, Jake Fevold, using an early version of our principled-design methodology, produced a paper, [P2754R0], providing a tabular comparison of the two approaches and immediately demonstrating that, in each individual dimension of interest, Köppe's erroneous-behavior-based solution was just as good or better than Bastien's defined-behavior solution. Importantly, Köppe's solution avoided the serious drawbacks that, for many, made Bastien's solution highly problematic.

Fevold's comparison paper, using early principled-design concepts, proved dispositive. Köppe subsequently produced a series of papers, [P2795R0]–[**P2795R4?**], proposing EB as a new form of standard-defined behavior and used this new behavior category to formalize his improved solution to this security problem. Köppe's paper reprinted Fevold's exemplary tabular analysis as conclusive evidence of the solution's superiority. Köppe's manifestly improved version of Bastien's immensely valuable "opening bid"[3] was adopted by EWG in Varna (c. June 2023).

The example we present here is an application of the now more refined *principled-design* approach (as described in "Writing a Principled-Design-Based Standards Proposal") to the original uninitialized-read security-vulnerability problem and Bastien's initial solution based on *defined behavior*. We then deduce the need for some modification to Bastien's "opening bid"; add to the analysis Köppe's augmented solution, based on *erroneous behavior*; and reach the same conclusions that transpired throughout late 2022 and the first half of 2023 but with more confidence, speed, and transparency.

[3]Bastien's revised version — [P2723R1] (January 2023) — of his original paper with its survey of viable alternatives and terse discussion of their pros and cons; Fevold's tabular representation; and Köppe's new solution all provided inspiration for our formalization of this *principled-design*-based Standards proposal process with its motivating principles and concerns.

6.2 Problem Statement

Roughly 10% of all security vulnerabilities in C++ can be traced back to reading (i.e., *lvalue-to-rvalue conversion*) an indeterminate value from an uninitialized nonclass (automatic) variable on the program stack, resulting in undefined behavior (UB).

6.2.1 Illustrative Example

```
void set1(int)     {         }  // does not touch argument in body
void set2(int& x) { x = 1; }  // Write only `x`.
void set3(int& x) { ++x;    }  // Read `x`, modify it, then write it.
extern void set4(int& x);      // Body is not visible in this TU.

int main() {

  int i, j, k;
  set1(i); // undefined behavior: _prvalue_ to _glvalue_ conversion
  set2(i); // Fine: i is set to 1.
  set3(j); // undefined behavior: reading an indeterminate value
  set4(k); // cannot know locally if behavior is undefined
  return i+j+k;
}
```

Notice that just passing an uninitialized variable to a function, e.g., `set1`, is UB, even if that function's body does not reference it at all. On the other hand, passing an uninitialized value by pointer or reference might be UB depending on whether the function reads it before writing to it, e.g., `set2` and `set3`. Finally, note that the compiler might not have enough information locally, e.g., `set4`, (or at all, if based on runtime input) to know for sure whether a given function call will necessarily result in UB.[4]

6.2.2 Business Justification

Such security vulnerabilities are problematic because they adversely affect commercial and government users directly. Moreover, security vulnerabilities that result from what is often perceived as ubiquitous UB, which is frequently a consequence of nonexpert use of the C++ language, are becoming increasingly likely to spur laws, regulations, or policies banning the use of supposedly unsafe languages, i.e., C and (perhaps by association) C++, for certain kinds of (e.g., safety critical) applications.

6.2.3 Measures of Success

By default, a nonclass variable on the program stack will have an indeterminate value. Local-neighborhood (single-translation-unit) static analysis is capable of detecting some but not all attempts to read an indeterminate value. A successful solution will substantially reduce, if not eliminate entirely, the possibility of reading an uninitialized nonclass (automatic) variable on the program stack without adversely affecting any other relevant aspects pertaining to C++ software, e.g., development, maintenance, performance, correctness, and so on.

6.3 Probative Questions

Information deduced from discussions on the WG21 reflector was used to produce a thoughtfully curated set of probative questions that identify salient properties of the various proposed and alternative solutions.

1. Does the solution ensure that all code that does not invoke UB continues to exhibit the same behavior it did before (i.e., as if no change had occurred)?
2. Does the solution allow legacy production code that does on occasion invoke UB due to uninitialized memory reads but is otherwise believed to work as desired (a) to always continue to compile and (b) to behave exactly as it did before?

[4]See Section 4 of [P2754R0] for additional relevant code examples.

3. Does the solution add behaviors that would make well-defined (i.e., not a defect) any behaviors that were undefined before (for example does `int i;` within a function body go from meaning “uninitialized” to meaning “initialized to zero”)?
4. Does the quality of implementation of the solution play a role in what programs compile? For example, one compiler detects a read-before-initialized defect and reports the program is ill formed, and another compiler, not detecting the defect but unable to prove that it doesn’t occur, prophylactically initializes the value to something without otherwise affecting compilation.
5. Is reading from a variable without first explicitly initializing or writing to that variable still a defect?
   — If yes, do tools that are not compiler integrated, such as Valgrind, continue to be usable to detect (i.e., as program defects) reads of uninitialized stack variables (e.g., for pre-existing defects in legacy code)?
   — If no, what values are appropriate for pointer-to-member (and similar) types?
6. Does the solution allow for evolution to some other, better, perhaps longer-term solution (e.g., a simpler initialization model)?
7. Is there a way to opt out of the change thus return to the behavior prior to the adoption of the solution (a) per-variable, using an annotation (e.g., an attribute) in the source code or (b) globally, using a tooling option (e.g., a compiler flag)?

6.4 Solutions

6.4.1 Proposed Solution: Always Zero-Initialize

All uninitialized automatic-storage-duration nonclass variables are initialized to a specific value. Numerical types would be initialized to zero. The value for pointer types is an open question. Note that many popular compilers already offer an option to zero-initialize:

```
void int f()
{
    int i;            // same as int i = 0;
    assert(0 == i);   // well defined and always true
}
```

In the function `f` above, `i` is always initialized to zero.

— Motivating Principles
  — By default, reading a variable of nonclass type cannot possibly result in UB.
  — The behavior of any previously defect-free code, particularly code that never attempts to read uninitialized variables of nonclass type defined on the program stack, is unchanged.
  — All code that previously compiled successfully continues to compile successfully.
  — Zero values are deterministic, stable, and perhaps more likely to be correct than indeterminate ones in production use.
  — The solution has been implemented (a) at all, (b) in actual use, and (c) at scale.
  — The average incremental runtime performance decrease has been demonstrated to be negligible or acceptable.
  — Whether a program will compile remains unchanged and portable; that is, the notion of what makes a TU ill formed is not further tied to the QoI of a particular compiler.
  — Build modes exist in which even a defective program can be made to revert to its previous (undefined) behavior.

— Concerns
  — By making previously undefined behavior well defined, we are no longer able to automatically diagnose such errors of omission. That is, many likely defects that the compiler might have reliably warned against previously would no longer be considered defects, and thus the compiler could no longer reliably warn against them.
  — The forced stability of the value makes knowing whether a potential correctness defect is present more difficult. Hence, we have effectively eliminated a security vulnerability, but we have lost the ability to readily detect the corresponding correctness errors (especially including those in legacy code).

    - While the typical runtime overhead has been demonstrated (with supporting benchmarks) to be negligible and occasionally even negative, such is not guaranteed to be the case in all situations, especially in those cases in which we are creating a large array on the stack to act as a raw memory buffer, such as local arenas used to support custom local memory allocators.
    - By adopting this solution in which previously undefined behavior is not fully defined, we might close the door on other, better solutions by making them no longer sufficiently backward compatible to the current semantics of the language.

6.4.2 Alternate Solution: Diagnose or Else Zero-Initialize

An implementation is permitted but not required to reject code that, if executed, would necessarily result in accessing an uninitialized nonclass automatic variable. Otherwise, if an implementation is unable to prove that no such access will occur, it is required to zero-initialize the variable and accept the code as well formed:

```
extern void g(int *p);  // `g` is defined elsewhere and might try to read `*p`.
void int f()
{
    int x;
    int y = x;      // allowed (but not required) to be ill formed

    assert(0 == x); // If not ill formed, `x` must be zero initialized.

    g(&x);          // `x` used before init? `g` in some other TU
                    // allowed (but not required) to be ill formed
}
```

In the function `f` above, either reading `x` is ill formed or `x` is zero initialized. We cannot know what `g` does, so it has to be initialized unless there is some form of whole-program optimization.

- Motivating Principles
    - Where practicable, make erroneous constructs ill formed, as opposed to undefined behavior, to enable catching such defects at compile time, which is clearly far less costly than catching them later.
- Concerns
    - Allowing QoI to determine whether a program compiles means that a program that compiles and works as intended on one compiler might fail to compile on another. Consider that a function that is called might easily be diagnosed as reading a nonclass automatic variable, but if that function is linked into a program but never called in practice, the program can be reasonably considered to be correct and yet be rejected by an aggressive compiler.
    - Ensuring consistency across compilers would mean defining precisely what is and isn't allowed to be diagnosed as ill formed, and just the effort to do that is prohibitive for Standards members and compiler vendors alike.

6.4.3 Alternate Solution: Require Initialization or Annotation in Source

All uninitialized automatic-storage-duration nonclass variables become ill formed. They must either be initialized explicitly or annotated as being explicitly uninitialized (e.g., using an attribute):

```
void int f()
{
    int x;                    // ill formed
    int y = 0;                // OK
    int z [[unitialized]];  // OK, some yet-to-be-defined annotation
}
```

In the function `f` above, we must either explicitly initialize the nonclass stack variable, (e.g., `y`), or explicitly annotate that it is not initialized (e.g., `z`); otherwise, it's ill formed (e.g., `x`). Note that any delayed form of initialization would require use of `std::optional` or a similar mechanism.

- Motivating Principles
  - Require users to make intent for an uninitialized nonclass automatic variable explicit in the code, thereby reducing inadvertent security vulnerabilities and improving maintainability.
  - Any *new* code must have a clear intent and must, by default, be safe (e.g., with regard to potential security vulnerabilities).
- Concerns
  - Existing code that does not initialize variables will not compile.
  - To disable errors, clients might choose to disable this feature wholesale by either (1) employing an implementation-defined compiler switch or, worse, (2) applying a script to gracelessly modify the source code to initialize (or otherwise annotate) every instance of an uninitialized nonclass automatic variable. This latter approach would have the effect of masking original intent and forever eliminating our ability to detect mechanically (e.g., through static analysis) correctness errors in legacy code.

6.4.4 Alternate Solution: Implementation-Defined Values But Reading Is Still UB

Each implementation is required to ensure that every uninitialized automatic-storage-duration nonclass variable is initialized to an implementation-defined value (unless the compiler can prove at compile time that said variable will not be read before being written); however, attempting to read that variable is nonetheless still considered UB (meaning that reading it is still considered a defect). One could, in development and testing, inject values that are likely to cause noticeable failures (e.g., signaling NaN, unaligned pointer, etc.) and, in production, inject best-guess and/or highly runtime efficient values, such as zero for integers:

```
void int f()
{
    int x;      // `x` has a non-indeterminate but otherwise unspecified value.
    int y = x;  // Attempting to read `x` is still UB.
}
```

In the function `f` above, the compiler is required to put some unspecified value in `x` to avoid "leaking" a previous value that just happened to be on the program stack or in a register. The assignment to `y`, however, remains UB (a defect).

- Motivating Principles
  - Enable effective correctness testing.
  - Correctness defects are in no way hidden, e.g., in legacy code.
  - Discourage users from relying (even accidentally) on particular values found in uninitialized (nonclass) automatic variables.
  - The change to C++ behavior is purely additive with no change to defined behavior; i.e., this solution can be applied surgically if and as needed without affecting backward (or forward) compatibility.
- Concerns
  - This solution as stated is not viable (i.e., cannot be represented in the current Standard by the current virtual machine). Undefined behavior (UB) has a specific meaning: *There are no requirements on*

that behavior. Hence, declaring behavior to be *undefined* and requiring that behavior to have some certain properties when invoked is inconsistent with the definition of UB.

6.4.5 Alternate Solution: Change C++ To Have Only Value-Initialization

Remove default initialization from C++ entirely and have only value initialization. The state of initialization in C++ is already very complex, and the cost of this complexity is dubious for the level of utility it affords. The entire initialization system could be pared down to one single form of initialization that provides values in all cases. This more fundamental change addresses uninitialized-variable problems as well as many other known problematic issues with initialization in general:

```
void int f()
{
    int x;           // Error
    int y(5);        // OK
    int z = INT_MIN; // OK
}
```

In the function `f` above, there is no support for default initialization. Any attempt to default initialize will result in an ill-formed program.

- Motivating Principles
    - Minimize overall complexity of the C++ language in the long term.
- Concerns
    - No clear concrete proposal for such a major revision of C++ has been offered nor is likely to be soon. Some consider the security concerns being addressed as distinctly urgent, so waiting for a broader-scope solution might be considered unacceptable.
    - Without a concrete proposal, we have no precise or credible way to evaluate how the proposed solution might compare with other solutions that are within scope for C++26.
    - Separately, this perhaps theoretically desirable simplification of initialization might well result in unacceptable performance penalties for a large swath of legacy code.

6.5 Curated, Refined, Characterized, and Ranked Principles

We've refined the original motivating principles, preserving the order in which they are described in "Solutions." 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 9: Importance and Objectivity of Principles

Rank	i	o	Principle ID	Principle Statement
6	9	@	safeDefault	By default, reading a variable of nonclass type cannot possibly result in UB.
1	@	@	sameCorrectBehavior	The behavior of *any* previously defect-free code, particularly code that never attempts to read uninitialized (automatic) variables of nonclass type, is unchanged. (Ignore dead code that might no longer compile.)
3	9	@	sameCodeCompiles	All code that previously compiled successfully continues to compile successfully.
15	1	5	betterProductionValues	Values used erroneously are deterministic, stable, and plausibly more likely to be correct than indeterminate ones (e.g., when used in production).
13	5	5	solidImpExperience	Implementation experience exists (a) at all, (b) in actual use, and (c) at scale.
8	9	5	zeroRuntimeImpact	Both the average and maximum incremental runtime performance decrease has been demonstrated in some way or otherwise proven to be zero (ideally) or else negligible.
4	9	@	illFormednessNotQOI	Whether a program will compile is guaranteed to be portable; that is, the notion of what makes a TU ill formed is not tied to the QoI of a particular compiler.
9	5	@	optOutBuildModes	Build modes exist in which even defective programs can be made to revert to what was their previous (undefined) behavior. (E.g., the status-quo solution has this mode by default.)
14	1	@	detectsDefAtCompileTime	Maximize the extent to which constructs that were previously UB are made ill formed.
17	-	@	eliminatesUnsafeDefaults	Makes ill formed any uninitialized nonclass automatic variables that are not annotated explicitly.
10	5	@	preventsUnsafeDefaults	Doesn't allow *new* (i.e., that which would be either undefined or ill formed now) uninitialized nonclass automatic variables that are not explicitly annotated as such to compile (not clear how this would be implemented).
7	9	@	facilRuntimeTesting	Enables effective correctness testing by allowing implementations to supply multiple (e.g., user supplied) deterministic values for uninitialized nonclass automatic variables.
5	9	@	doesNotMaskDefects	In no way hides correctness defects, e.g., in legacy code. (Also consider foreseeable indirect consequences.)
12	5	5	discouragesUseOfDefaults	Discourages users from relying (even accidentally) on uninitialized nonclass automatic variables having a particular value.
11	5	@	bidirectCompatible	The change to C++ behavior is purely additive with no change to fully defined behavior, ensuring complete bidirectional semantic compatibility (i.e., either is substitutable for the other).
16	1	-	reducesLangComplexity	Helps to reduce or minimize the overall complexity of the C++ language.
2	@	5	isViable	There is no reason to believe that the solution is not implementable.

6.6 Compliance Table: Solutions Scored Against Ordered Principles

NB: To avoid repeating the compliance table, we add a new solution, G, here, and we describe solution G after the table analysis.

Col		**Solutions**
A.	Status-Quo Default	**No Change to the Standard**
B.	Original Proposed	**Always Zero-Initialize**
C.	Original Alternate	**Diagnose or Else Zero-Initialize**
D.	Original Alternate	**Require Initialization or Annotation in Source**
E.	Original Alternate	**Imp-Defined Values but Reading Is Still UB**
F.	Original Alternate	**Change C++ To Have Value-Initialization Only**
G.	Newly Proposed	**New Category: Erroneous behavior**

Table 11: Compliance Table

Rank	**i**	**o**	**Principle ID**	**A**	**B**	**C**	**D**	**E**	**F**	**G**
1	@	@	sameCorrectBehavior	@	@	@	@	@	@	@
2	@	5	isViable	@	@	@	@	-	@	@
3	9	@	sameCodeCompiles	@	@	7	1	@	?	@
4	9	@	illFormednessNotQOI	@	@	1	@	@	@	@
5	9	@	doesNotMaskDefects	@	-	@	9	@	5	@
6	9	@	safeDefault	-	@	@	@	@	@	@
7	9	@	facilRuntimeTesting	-	-	-	-	5	-	@
8	9	5	zeroRuntimeImpact	@	7	8	9	7	?	7
9	5	@	optOutBuildModes	@	@	@	-	@	-	@
10	5	@	preventsUnsafeDefaults	-	-	5	@	-	@	-
11	5	@	bidirectCompatible	@	-	-	-	@	-	@
12	5	5	discouragesUseOfDefaults	7	-	5	@	3	-	5
13	5	5	solidImpExperience	@	9	-	7	-	-	5
14	1	@	detectsDefAtCompileTime	5	-	7	9	-	-	5
15	1	5	betterProductionValues	-	5	5	5	7	5	9
16	1	-	reducesLangComplexity	-	3	-	3	-	9	-
17	-	@	eliminatesUnsafeDefaults	-	-	3	9	-	-	@

6.7 Analysis of the Compliance Table

Row 1 — A B C D E F G — sameCorrectBehavior(@,@)

All of the solutions scored @ for this first principle, so row 1 offers no discriminating information.

Row 2 — A B C D E F G — isViable(@,5)

Every solution except E scored @ against the second principle; E scored -. E is now eliminated unless new information arises.

Row 3 — A B C D _ F G — sameCodeCompiles(9,@)

This third principle effectively discriminates between the six remaining solutions. Three solutions, A, B, and G, score @, and C, D and F fall short.

Alternate solution C, because it correctly diagnoses what would be UB if executed in (possibly even dead) code doesn't fully satisfy (7) this principle as stated, leaving this alternate solution in danger of elimination. (In the next row's analysis, we'll set a lowercase c to indicate C's diminished status.) Alternate solution D, on the other hand, is typically going to cause even perfectly correct programs having no dead code to be reworded to conform to these new, much more austere rules and hence fails nearly completely (1) to satisfy this requirement, effectively removing it from the list of viable candidates. Finally, alternate solution F is not currently specified, so we cannot know (?) if and to what extent this solution might eventually comply with this principle, eliminating it as a viable contender, at least for C++26.

Row 4 — A B c _ _ _ G — illFormednessNotQOI(9,@)

Our fourth principle leaves our front-running solutions A, B, and G in good shape, each scoring @, but eliminates alternate solution C, which essentially fails (1) to satisfy this principle because, without expensive and detailed specification, it will lead to a loss in portability with respect to compilation.

Rows 5–6 — A B _ _ _ _ G — doesNotMaskDefects(9,@) and safeDefault(9,@)

In these two critically important rows, *principled design* distinguishes itself from other methodologies (or the lack thereof) by laying bare the quintessential issue at hand: Is the original proposed solution B really better than the default status-quo solution A or not, and why? We have determined that doesNotMaskDefects has the same importance and objectivity score but is slightly more important than safeDefault in row 6, which would imply that the status-quo solution A is a favorite to be preferred over solution B. On issues of this kind, opinions will vary, at which point we need to consider other principles of lesser weight in the column. When we are not sure which way to lean, we have a standard principle (a slightly stronger version of which can be found row 11) that states that, if there is a choice between two solutions, prefer one that allows for backward-compatible migration to another that doesn't allow such a migration. Using this standard principle in conjunction with these two principles of near equal weight, there's little doubt that the prudent decision would be to hold off on adopting B and look for a better solution that doesn't give up on our fifth or sixth most important principle. Note that our, as-yet-to-be-discussed, new solution G has yet to receive anything less than a perfect score.

Row 7 — a b _ _ _ _ G — facilRuntimeTesting(9,@)

We see that the status-quo solution A and the proposed solution B do nothing at all to help with diagnosing correctness bugs via runtime testing. Reasonable people may disagree as to the relative importance of this principle, yet a solution that admits such testing cannot reasonably be dismissed as no better than one that doesn't. (At this point, G seems to be the winner, but we'll retain A and B a bit longer to be absolutely certain.)

Rows 8–12 — a b _ _ _ _ G

For ease of reading, the scores for solutions A, B, and G against these principles are repeated here.

Rank	i	o	Principle ID	a	b	G
8	9	5	zeroRuntimeImpact	@	7	7
9	5	@	optOutBuildModes	@	@	@
10	5	@	preventsUnsafeDefaults	-	-	-
11	5	@	bidirectCompatible	@	-	@
12	5	5	discouragesUseOfDefaults	7	-	5

The next five principles span a fairly wide range of weights. All but one of the principles actively favor the status-quo solution A (with that remaining case being a tie) over the originally proposed solution B. The new solution, G, continues to score well, especially after achieving a perfect score on the first seven principles. Note that the three False scores for B on rows 10–12 are sufficient to eliminate it.

Rows 13 — a _ _ _ _ _ G — solidImpExperience(5,5)

Here we have another principle with a controversial relative weighting. For some, not aggressively satisfying this principle might be a nonstarter, and for others, knowing that it can be done might be good enough. Although solution B is eliminated, we know that solution G is, in spirit, very similar to solution B, which did strongly satisfy (9) this principle. Hence, we state that solution G moderately satisfies (5) this principle though G does require us to define and adopt an entirely new class of standard behavior, namely *erroneous behavior* (EB).

Rows 14–16 — a _ _ _ _ _ G

For ease of reading, the scores for solutions A and G against these principles are repeated here.

Rank	i	o	Principle ID	a	G
14	1	@	detectsDefAtCompileTime	5	5
15	1	5	betterProductionValues	-	9
16	1	-	reducesLangComplexity	-	-

Each of these principles was determined to be of low importance; hence, almost nothing here will sway our previous decisions. That said, new solution G is strictly better than status-quo solution A.

Rows 17 — a _ _ _ _ _ G — eliminatesUnsafeDefaults(-,@)

The final principle was determined to be entirely irrelevant (-) and hence offers no useful information, yet new solution G is better than the status-quo solution A.

From the above, we conclude that the only viable change to the Standard would be the new solution G presented below.

6.8 A Better Solution – New Behavior Category: Erroneous Behavior

All uninitialized automatic-storage-duration nonclass variables are initialized to an implementation-defined value, but reading that value is *always* considered a defect (like *UB*) yet still defined (unlike *UB*). The term proposed by Thomas Köppe proposed the new term *erroneous behavior* for this defined but undesirable behavior. Any program that contains *EB* is incorrect, but the behavior is *implementation defined*. Some applications could treat *EB* as UB for performance, use hostile default values for testing and development, or use somewhat safe and/or default values for production. New opportunities for tools to detect correctness bugs become available as well. Note that once we standardize this new kind of behavior, we can then apply EB to other situations that are currently UB, e.g., deleting a nontrivially destructible object having an incomplete type or relying on a particular order of evaluation of unsequenced subexpressions having side effects.

— Motivating Principles
  - — The solution is viable, meaning that it can be implemented with suitable work to extend the Standard.
  - — Discourage users from relying (even accidentally) on particular values found in uninitialized nonclass automatic variables.
  - — Do not hide correctness defects, e.g., in legacy code.
  - — Avoid affecting the compilability of existing code (e.g., no QoI issue).
  - — Do not affect the runtime meaning of existing code.
  - — Avoid affecting the observable behavior of correct programs.
  - — Have no direct effect on any behavior for even a defective program as long as the program does not attempt to read an uninitialized nonclass automatic-storage-duration variable.
  - — Build modes will exist in which even a defective program can be made to revert to what was its previous (undefined) behavior.
  - — The change to C++ is purely additive with no effect to defined behavior; i.e., the solution can be applied surgically if and as needed without affecting forward or backward compatibility.
  - — Enable effective correctness testing.

NB: Principles 4 through 7 were not in our original solutions-and-motivating-principles matrix. Since these principles are unlikely to eliminate this solution and since it has already shown itself to be the optimal solution, we have no reason to redo the analysis.

— Concerns
  - — Programs that exhibit erroneous behavior in a new compiler release but that previously behaved as desired might no longer behave as expected.
  - — The Standards Committee will need to adopt a new behavior, EB.

6.9 Recommendations

Based on the analysis above, we feel comfortable presenting only two alternatives. Our strong recommendation would be to adopt solution G, *New Behavior Category: Erroneous Behavior*. Otherwise, we would recommend the status-quo default solution A. Any other even remotely viable solution considered runs the risk of precluding better, backward-compatible solutions in the future.

7 Conducting a Principled-Design-Aware Standards Meeting

So far, we have assumed that a single author or group of authors is preparing the proposal, providing all the analysis, and presenting a recommendation, asking only for the greater C++ Standards community to audit and approve it as is. While that's the ideal case, not everyone involved in submitting proposals is necessarily going to adopt this approach or have the same opinion.

First, not everyone who writes a proposal has necessarily researched the history of the problem, gathered all possible alternate solutions, provided all the related motivating principles, and carefully considered all the associated concerns. Hence, even a paper that purports to use our principled-design approach might fall short of being compelling to a larger group, especially if other, well-known competing solutions do not appear in the analysis.

Second, multiple contemporaneous competing papers might propose distinct incompatible solutions, motivated by competing principles that are evaluated subjectively to have differing priorities. Occasionally, one solution will ignore the motivating principles of another solution either out of honest oversight or because they are deemed irrelevant. In rare situations, a particular property of a solution will be deemed by some as a positive and by others a negative. For example, the attribute-like syntax for the MVP of the new C++ Contracts facility suggested that a contract-assertion annotation (CAA) could be thought of as optional; that position was considered a definite plus to some (these authors included), while others felt it was a clear minus. In the end, other, more important principles — such as lightweight syntax — won.

Whether the respective principles are valued at disparate priorities or are simply overlooked, a paper that doesn't properly address all the relevant issues will require additional work, which is exactly the kind of oversight we rely on from a larger group of our peers. We need other experts to assess and provide guidance based on the principles and insights they might contribute.

In fact, the notion of principled design was used effectively at the SG21 telecon on February 1, 2024. Polarized subgroups were proposing solutions that had no hope of gaining consensus and were thereby risking the Contracts MVP's inclusion with C++26. Using his own simplified adaptation of our principled-design methodology [P3113R0], Timur Doumler, the assistant SG21 Chair, walked attendees through the requirements until Lisa Lippincott resurrected a previously discarded solution that gained strong consensus. This solution was no one's first or even second choice but achieved strong consensus (SF = 7, F = 5, N = 1, A = 1, SA = 0) because it (1) provided all of what was needed (and most of what was desired) for the MVP, (2) didn't violate anyone's principles, and (3) allowed space for a better future solution with backward compatibility.

In this section, we relate our experience working in a small group to reach consensus on the scoring of principles. We generalize that experience by proposing an approach to reconciling, in a larger group, one or more papers containing solutions to a given problem. In this way, we hope to streamline the process when a subgroup of the C++ Standards Committee convenes. The subgroup will provide any needed arbitration with respect to characterizing principles and evaluating the compliance of solutions to said principles as efficiently as practicable.

Proposals' adherence to the principled-design process will, we hope, alleviate most debate and thus conserve valuable Committee time.

7.1 Agreement by Acclamation (Unanimous Consent)

In a typical C++ Standards meeting, those who are fully aware of all the nuance and detail of an issue often have little disagreement. Participants who only partially understand the issue, however, understandably raise questions, and the ensuing discussion and debate consumes scarce Committee time.

Ideally, the author of each Standards proposal will have taken the time to research the issue, capture any plausible alternate solutions along with their uniquely compelling motivating principles (and related concerns), and fairly represent those competing solutions along with their own solution. The more alternative solutions provided and the more relevant principles considered, the more likely the proposal's principled-design analysis will be sufficiently comprehensive to be dispositive.

Given a single proposal, the role of the Committee is to first review the proposed and alternate solutions to ensure they and their appropriate motivating principles and concerns have been adequately described. Next, each refined principle is reviewed for clarity and to confirm that no one objects strongly to the characterizations of objectivity or importance assigned to them. If objections arise, Committee discussion will be necessary. If a change is made, re-evaluation of the principle ranking might be needed. A change in ranking doesn't affect the scoring of the solutions relative to that principle, but it might affect the final recommendation.

7.1.1 Delegating the Initial Evaluation

When more than one proposal on a topic is to be considered or when one or more proposals on the same topic has not followed (or has insufficiently followed) the principled-design process, the Committee will typically not have a single sequence of solutions along with their motivating principles and concerns nor one set of unique refined principles to be applied to score the collected proposed and alternate solutions.

For efficiency, the Committee, rather than trying to perform this task in real time, would be wise to request that the proposals' authors work together to produce a unified paper having both recommended solutions (along with their motivating principles and concerns), all alternate solutions (again, with their motivating principles and concerns), and a single sequence of refined principles to be used to score the solutions.

The coauthors then use the ranked principles to score the solutions and provide some collective analysis and recommendation to the Committee. Even if no clearly optimal solution arises, we still have achieved a simple and effective process for auditing what was done and discussing, based on the results, which, if any, of the proposed solutions is to be preferred over the status quo.

7.2 Gathering Aggregate Input from the Group

Under some circumstances, the Committee subgroup may choose to characterize, order, and apply the motivating and other relevant principles collaboratively in real time. This process, though effective, can be slow, especially when scoring multiple alternate solutions having many relevant principles. We provide some effective group-oriented techniques to increase efficiency.

7.2.1 The Importance of Coarse Categorizing

Recall that we limited the available scores for importance (@ 9 5 1 -) and especially objectivity (@ 5 -). This restriction is the result of practicing the methodology and discovering that consensus is better served when the precision with which an answer can be expressed is not allowed to exceed the accuracy with which the number expressed represents the true value.

For importance, a principle can be imperative (@); it can be of high, medium, or low importance; or it can be irrelevant (-). Any further precision increases the risk of unnecessary disagreement. Moreover, at finer levels of granularity side-by-side comparison is often needed to produce the final ordering anyway, which is why, after ranking, a final pass is made to ensure that the order is what we intuitively want it to be, and the default (importance, objectivity) ordering can be overridden (e.g., by consensus vote) if need be. Finally, if we need to ask the group for input, having five possibilities meshes nicely with the familiar five-way poll (SF F N A SA), and we'll discuss that more later.

A principle is either provably objective (@), somewhat objective (5), or subjective (-). Most people can agree about when a principle is provably objective, so the only question remaining is whether the principle is somewhat objective (5) or subjective (-). Any more gradations on this category would overly complicate the process of gaining consensus. Absent unanimous consent, a simple show of hands is typically all that's needed.

Scoring compliance requires more precision. Experience shows, however, that restricting the choices to True, the percentages 90%, 70%, 50%, 30%, and 10%, and False along with the out-of-band value Unknowable (?), meaning that an answer is not currently easily knowable, is typically enough (@ 9 7 5 3 1 -) to convey an appropriate level of compliance. In rare cases, the interstitial values 80%, 60%, 40%, and 20% (8 6 4 2) can be pressed will be needed.

7.2.2 Characterizing Principles: RICO

When an author writes a principled-design-based proposal paper, the task is straightforward. The author simply does their best to provide all the motivating and other relevant principles and the related concerns; concisely state, characterize, and rank them; and apply them to the solutions considered in the paper. The values for importance, objectivity, rank, and compliance are whatever the author says they are. Even when a paper has multiple authors, characterizing principles and scoring solutions is usually easy because the discussion is among a small group of presumably like-minded individuals.

A wider group, however, typically has a greater diversity of experience and opinion. Using principled design, we first focus the larger group not on the suggested solutions, but on the ordered suite of relevant[5] principles that will be ultimately lead us to an optimal solution, perhaps one that isn't even among those that have already been considered.

Relevance: When we consider a principle, we want to quickly gauge whether it is *relevant*. Relevance is binary. If the principle has no relevance, we have no reason to characterize it further, but if one person thinks it does or might have relevance, then the principle is worth some discussion.

Consensus: For a single proposal, we presume 100% of the authors agree; if that's not the case, then perhaps the authors need to have more discussion. In a meeting, however, Committee members will disagree, and we'll need to find characterizations that we all can accept. We, therefore, consider the weight or relative priority of a principle in a group to be affected by two measures: (1) consistency, the fraction of the group who consider a given principle relevant and (2) median importance, the median value of the respective importance values taken from only those individuals who consider the principle relevant.

Hence, the acronym *RICO* stands *Relevance*, *Importance*, *Consensus*, and *Objectivity*.

— *Relevance*: At least one person in the room thinks the principle might be relevant.
— *Importance*: Median value among those who believe the principle is relevant.
— *Consensus*: Fraction of the group who think the principle is relevant and median importance value given by those who find the principle relevant.
— *Objectivity*: Very strong consensus (> 80%) on provable (@) or a majority between somewhat objective (5) or subjective (-), with ties rounding up.

7.2.2.1 Using Five-Way Polls Effectively to Collect RICO Data

By taking a single five-way poll, we can typically derive the relevance, importance, and consensus from the poll (with objectivity typically confirmed with a simple show of hands).

The mapping is straightforward.

Value	Vote
@	SF
9	F
5	N
1	A
-	SA

[5]An author might call out a principle that was originally thought to be relevant but later proved to be otherwise.

As an example, consider this poll result for determining relevance, importance, and consensus at once.

SF/@	F/9	N/5	A/1	SA/-
0	4	5	1	0

This poll would lead us to the following conclusion.

Measure	Value	Reasoning
Relevance	True	something other than SA was nonzero
Importance	5	median of the scores excluding SA
Consensus	1.0	10/10 of the votes were not SA

In rare cases, someone will feel that a principle is *reversed*, meaning that the *negation* of the principle is relevant and of some importance. That person will vote SA, but after the poll, the chair will ask for a show of hands to determine if and how many voted reverse, and that person will raise their hand.

As a second example, let's now consider this poll result

SF/@	F/9	N/5	A/1	SA/-
0	2	3	2	3

Measure	Value	Reasoning
Relevance	True	something other than SA was nonzero
Importance	5	median (rounded up) of the scores excluding SA
Consensus	0.7	7/10 of the votes were not SA

The chair, seeing one or more SA votes, asks for a show of hands that the principle is reversed, and one person responds, meaning that this person favors a principle in which the opposite property is desirable. The poll results are then amended to reflect this finding of the group.

SF/@	F/9	N/5	A/1	SA/-
0	2	3	2	2/1 — two irrelevant, 1 opposite

A low consensus can be used to help us break ties for principles with otherwise equal importance scores, but see the figure (in particular, where consensus is low but importance is high) in the next section.

Again, a principle is scored as provably objective (typically by unanimous consent) or as somewhat objective or subjective by a majority vote (rounded up). Obtaining agreement should be easy enough in most cases.

7.2.2.2 Polls That Necessitate Further Discussion

Most of the time, we would expect that, after a poll is taken, some quick discussion of the (median) importance value can occur before the value is entered into its rightful place in the importance and objectivity table, ideally along with the straw-poll vote for posterity. Certain patterns, however, must be considered as alarms that something is amiss. Let's consider four scenarios reflecting four quadrants of the importance and consensus matrix.

```
     @  +---+---+---+---+---+---+---+---+---+---+
  I     |                   :                   |
  m     |  Low Consensus    :   High Consensus  |
  p  9  + High Importance   :   High Importance +
  o     |                   :                   |
  r     |               IV. : III.              |
  t  5  + - - - - - - - - - + - - - - - - - - - +
  a     |                I. : II.               |
  n     |                   :                   |
  c  1  + Low Consensus     :    High Consensus +
  e     | Low Importance    :    Low Importance |
        |                   :                   |
     -  +---+---+---+---+---+---+---+---+---+---+
        0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
                          Consensus
```

I. Low Consensus; Low Importance

SF/@	F/9	N/5	A/1	SA/-
0	0	1	1	8

This principle (ranked among the lowest of the 5s) will likely have little impact on the selection of the final solution. Note it and move on.

II. High Consensus; Low Importance

SF/@	F/9	N/5	A/1	SA/-
0	0	3	7	0

The groups agrees that this principle (ranked among the highest of the 1s) is not especially important. Note it and move on.

III. High Consensus; High Importance

SF/@	F/9	N/5	A/1	SA/-
3	7	0	0	0

The group agrees that this principle (ranked among the highest of the 9s) is very important, but 3 of 10 felt this principle was imperative. In such cases, this principle might have broad implications, and violating it even once could make something that has previously been 100% True in the Standard no longer so. In cases like this, we should probably further discuss the principle to allow the three strongly-in-favor voters the opportunity to win over some of the seven in-favor voters (or vice versa).

IV. Low Consensus; High Importance

SF/@	F/9	N/5	A/1	SA/-
0	3	0	0	7

Arguably, the worst case occurs when a few people feel the principle is extremely important (perhaps some even think it's imperative) and all the others think it's of no importance at all. In this example, either three people see something that the other seven do not or what three people feel they need in their code is perceived as unnecessary by the other 7.

This result is highly inconclusive because the three people needing the principle might be implementers or advanced users who are facilitating new features that will be used by retail customers moving forward. We must also be reminded (see [stroustrup94]) of the important design principle stating that supporting (fully) a useful feature (to some) is better than preventing all misuses of that feature (by everyone). In any event, this sort of dichotomy between high and no (or even low) importance means that significantly more discussion is mandatory.

7.2.3 Assessing the Compliance of Solutions to Principles

Our best advice is to task one person or a small group to assess, outside the meeting, the compliance of the solutions to the principles and come back to the large group with a proposal. The Committee members can then efficiently audit the compliance table (again ideally outside the meeting) and reconvene to discuss any objections. If everyone agrees enough with the answers such that a small change in the calculus will have no affect on the decision, then the proposal can be adopted as is by unanimous consent.

7.2.3.1 Using Five-Way Polls to Collect Aggregate Opinion

Whether such an approach would be cost effective is unclear. We can take an N-way poll around some hypothetical mean and see which number has the most consensus. Alternatively, we can simply have a vote for who likes or can accept each value (@ 9 7 5 3 1 - ?). More experimental data and experience is needed.

7.2.3.2 Electronic Polling

Electronic polling equipment that allows a "yes" or "no" vote or an N-way poll is one possibility for collecting and scoring data much more quickly. A factor of 10 improvement would likely qualitatively change the standard operating procedure.

7.3 Summary

The diversity of the Committee is one of its greatest strengths, yet differences of opinion about the prioritization of design principles for competing solutions can lead to impasses. By first, *as a group*, establishing all relevant design principles and then collectively evaluating, prioritizing, and ordering each of these principles, we objectively and methodically leverage the diversity of the group and focus its decision making via a *shared* understanding of the requirements and objectives. This cumulative effort can then be captured in an easy-to-understand format, and each Committee member can ultimately voice their opinion via a familiar five-way vote (SF F N A SA). With this data, the Committee can then make a well-informed decision based on the agreed-upon design principles and prioritization and on a fair comparison of potential solutions.

8 Conclusion

Choosing, as a large group, between competing engineering solutions can be challenging. Principled-design-based proposals provide the Committee with a reasoned comparison of available solutions and a well-founded recommendation. Likewise, the principled-design-based process for meetings conserves scarce Committee time by focusing discussion on the *principles* that drive decisions, thus harnessing the benefits of diversity of experience and minimizing debate that doesn't serve decision making. This approach for determining optimal solutions based on ordered principles is both effective and efficient and would be valuable as an optional standard operating procedure throughout the various subgroups of the C++ Standards Committee (WG21).

9 Appendix: Worked Examples of Papers

This appendix contains examples of the principled-design process applied to Standards proposals, each considering competing viable solutions. In each example paper, we weigh several possible solutions and use our principled-design methodology to advocate for one or two of them. Each example could be separated into its own paper and, with a little more work, be a viable WG21 proposal. None of these examples, however, proposes wording; that work is beyond the scope of this paper.

10 Paper 1: Improving Syntax for Temporary Namespace Changes

10.1 Brief Historical Recap

Namespaces, templates, and template specializations have been a feature of C++ since the original 1998 Standard. Template specializations as a customization point were used more frequently as type traits, and similar techniques were popularized around the time that C++03 was published.

Opening and closing nested namespaces was simplified with the adoption of [N4230] for C++17, and while Tristan Brindle proposed a syntax to more easily specialize templates from a foreign namespace in [P0665R1], no progress has been made to simplify writing code for this occasional need.

10.2 Problem Statement

Occasionally, a developer needs to specialize templates from a completely different namespace hierarchy than the one in which their code resides, e.g., for template customization points like traits or hash functions among others. The specialization usually requires closing the current namespace(s), opening the namespace(s) where the template(s) needs to reside, then specializing the template(s), closing the entity namespace(s), and possibly reopening the previous namespace state. This formulation is needlessly verbose, and a more succinct alternative is desirable.

10.2.1 Illustrative Example

Let's consider an (abbreviated and amended) example in which the `std::hash<T>` customization point is specialized for type `Enterprise::Package::point` by specializing `::std::hash<T>` (see 22.10.19 [unord.hash]):

```
#include <functional>

namespace Enterprise::Package {

    struct point{int x; int y;};

} // close Enterprise::Package namespace

namespace std
{

    template<>
    struct hash<Enterprise::Package::point>
    {
        // implement `operator()`...
    };

} // close std namespace

namespace Enterprise::Package {
    // ... work continues here...
} // close Enterprise::Package namespace
```

Note that since `namespace std` is being reopened, inadvertently adding declarations or definitions rather than just specializations is possible and could lead to undefined behavior (see 16.4.5.2.1 [namespace.std]).

10.2.2 Business Justification

An easier way to define such specializations will improve code clarity by simplifying the defining of those specializations near their associated types without disrupting the logical flow of the declarations.

10.2.3 Measure of Success

A good solution should involve less typing, provide no less clarity, and limit the opportunity to misuse the new feature.

10.3 Probative Questions

Does reopening foreign namespaces carry risk?

Yes, name lookup rules will need to be updated to clarify whether the enclosing namespace scope is part of the lookup set and whether it is an associated namespace for the purposes of name lookup — both of which potentially make ADL even more complicated and confusing than it is now.

Do unrestricted additions to scoped reopened namespaces have associated costs?

Yes. Parsers for existing tools that are not compilers will have to be updated to find namespaces and their contents in foreign scopes; this could be challenging for the simplified parsers used in restricted contexts, such as syntax highlighters.

A number of design concerns must be addressed regarding name lookup and associated namespaces; this topic is a notoriously complex part of the Standard to revisit.

10.4 Solutions

10.4.1 Proposed Solution: Allow Specializing Fully Qualified (with Leading `::`) Templates

This solution was previously presented in [P0665R1], “Allowing Class Template Specializations in Associated Namespaces” by Tristan Brindle and was approved via vote in Rapperswil in 2018. The authors were unable to find, in a brief search, any further activity on this proposal.

In this approach, template specializations can be declared in situ without closing the current namespace or reopening the containing namespace by *fully* qualifying the template name, e.g., `::std::hash`, as long as one of the template arguments is related to the enclosing namespace:

```
#include <functional>

namespace Enterprise::Package {

    struct point{int x; int y;};

    // Note that we did not reopen the `std` namespace here. Instead, we
    // fully qualify the name of the `::std::hash` template we are
    // specializing.  We need not fully qualify `point` since entities
    // defined up to here in namespace `Enterprise::Package` are visible.

    template<>
    struct ::std::hash<point>
    {
        // implement `operator()`...
    };

    // Note that we cannot specialize
    // ::std::hash<Enterprise::OtherPackage::sometype> here since none of the
    // template arguments are related to Enterprise::Package.

    // ... work continues here...

} // close Enterprise::Package namespace
```

Note that the **full** qualification is possible but not required.

Had we been specializing `Enterprise::OtherPackage::trait`, the example would simply be:

```
#include <otherpackage_trait>

namespace Enterprise::Package {

    struct point{int x; int y;};

    // Note that we did not reopen the `OtherPackage` namespace here. Instead,
    // we sufficiently qualify the name of the
    // `::Enterprise::OtherPackage::trait` template we are specializing such
    // that we can find it from this sibling namespace.

    template<>
    struct OtherPackage::trait<point>: public std::true_type { };

    // ... work continues here...
} // close Enterprise::Package namespace
```

- Motivating Principles
  - C++ should become easier to use correctly and harder to use incorrectly.
  - Solutions should lead to easily maintainable code.
  - Solutions should lead to clear code.
  - A solution should not affect the language beyond the specific problem it is solving.
  - Prefer solutions already in progress toward standardization (all else being equal).
- Concerns
  - Limiting this solution to specializations on types in the enclosing namespace, as discussed in [P0665R1], might be necessary.

10.4.2 Alternative Solution: Allow Specializing Qualified Templates

In this approach, template specializations can be declared in situ without closing the current namespace or reopening the containing namespace by fully qualifying the template name, e.g., `std::hash`, as long as one of the template arguments is related to the enclosing namespace.

Note: This is similar to the "Allow specializing fully qualified (with leading `::`) templates" solution but would require fewer changes to allow a leading `::` on a specialization. In fact, the "Allow specializing fully qualified (with leading `::`) templates" solution simplifies to this case if the template being specialized is in the currently open namespace hierarchy.

```
#include <functional>

namespace Enterprise::Package {

    struct point{int x; int y;};

    // Note that we did not reopen the `std` namespace here. Instead, we
    // qualify the name of the `std::hash` template we are specializing.
    // We need not fully qualify `point` since entities defined up to here
    // in namespace `Enterprise::Package` are visible.

    template<>
    struct std::hash<point>
    {
        // implement `operator()`...
    };

    // Note that we cannot specialize
    // std::hash<Enterprise::OtherPackage::sometype> here since none of the
    // template arguments are related to Enterprise::Package.

    // ... work continues here...

} // close Enterprise::Package namespace
```

- Motivating Principles
  - Syntax is not overly constrained.
- Concerns
  - This solution could be ambiguous when specializing `C::D<>` if namespace `C` also exists in one of the currently open namespaces.

10.4.3 Alternative Solution: Allow Opening `::namespace`

In this solution, using `::` as a prefix in a namespace definition or nested namespace definition (9.8.2.1 [namespace.def.general]) would be possible and would allow us to create specializations without needing to close then reopen the namespace of the type on which we're specializing:

```
#include <functional>

namespace Enterprise::Package {

    struct point{int x; int y;};

    namespace ::std
    {

    template<>
    struct hash<Enterprise::Package::point>
    {
        // implement `operator()`...
    };

    } // close std namespace

    // ... work continues here...

} // close Enterprise::Package namespace
```

This solution maintains the property that definitions of entities within a given namespace are always defined within that namespace, which may simplify search logic for C++ related tools such as editors with code completion features.

- Motivating Principles
  - All members of a namespace are defined within that namespace.
- Concerns
  - Opening an existing namespace from within a different namespace will lead to new concerns in the already complex name-lookup rules.

10.5 Curated, Refined, Characterized, and Ranked Principles

We follow the process described by this paper.

1. Collect the principles underlying the design discussion.
2. Score the principles according to importance and objectivity, and rank them using those scores.
3. Score the proposed solutions against the principles.
4. Review the results, look for new solutions, and repeat from step 3 if necessary.

First we collect all the motivating principles from the solutions above, and copy them verbatim into the table below. Then we review the wording to ensure there is a clear and unambiguous objective statement for each principle. At this point, we might add to the bottom of the table some more general principles that are known to apply to many problems.

Next we score the principles on importance and objectivity and use the results to rank them. We can then proceed to scoring the solutions against the principles, using a measurement scheme in which higher is better, @ is True (100%), 9 is 90%, ..., 1 is 10%, and - is False (0%).

Table 24: Importance and Objectivity of Principles

Rank	**i**	**o**	**Principle ID**	**Principle Statement**
1	@	@	IsViable	There is no reason to believe that the solution is not implementable.
2	@	5	EaseOfUse	C++ should be easier to use correctly.
3	@	5	Maint	Solutions should lead to easily maintainable code.
4	@	5	Clear	Solutions should lead to clear code.
5	9	@	Limit	Do not affect unrelated code.
6	9	@	Approval	Prefer solutions already in progress.
8	5	5	Syntax	Syntax is not overly constrained.
7	5	@	Lexical	All members of a namespace are defined lexically within that namespace.
9	5	5	PrecSol	Do not preclude possibly better solutions.

10.6 The Compliance Table

Now we will score the solutions against the ranked principles. We introduce the status-quo solution as the current baseline that viable solutions must beat.

Solutions are represented as

A. Status Quo
B. Specialize `::namespace::type`
C. Specialize `namespace::type`
D. `namespace ::namespace { }`

The scale is @ means True (100%), 9 means 90%, ..., 1 means 10%, and - means False (0%).

Table 25: Compliance Table

Rank	i	o	Principle ID	A	B	C	D
1	@	@	IsViable	@	@	@	@
2	@	5	EaseOfUse	-	@	@	7
3	@	5	Maint	-	@	@	@
4	@	5	Clear	-	@	8	@
5	9	@	Limit	-	@	@	5
6	9	@	Approval	@	@	-	-
7	5	@	Lexical	@	1	3	@
8	5	5	Syntax	5	7	@	@
9	5	5	PrecSol	@	1	5	5

10.7 Analysis of the Compliance Table

Row 1 — A B C D — IsViable(@,@)

All of the proposed solutions appear equally viable.

Row 2 — A B D D — EaseOfUse(@,5)

All the proposed solutions — omitting A, of course, — are better than the status-quo solution. Solution D, allowing the opening of foreign namespaces, scores only 7 and is in danger of being eliminated compared to the other two solutions; we will, however, retain solution D for now (without looking ahead) in case subsequent imperative principles cause other solutions to fail. (In the next row's analysis, we'll set a lowercase d to indicate D's diminished status.)

Rows 3–4 — _ B C d

For ease of reading, the scores for solutions B, C, and D against these principles are repeated here.

Rank	i	o	Principle ID	B	C	d
3	@	5	Maint	@	@	@
4	@	5	Clear	@	8	@

The remaining imperative principles all have the same importance, and only solution C has less than perfect marks, scoring only 8 on the principle of producing clear code. We will demote solution C like we did for solution D and represent it by a lowercase letter for subsequent rows.

Row 5 — _ B c d — Limit(9,@)

Here we see that solution D scores only 5 on avoiding affecting unrelated code, so given its already diminished status, we eliminate it from further consideration. The other two solutions proceed with a perfect grade on this row.

Row 6 — _ B c _ — Approval(7,@)

Now we see that a proposal is already in progress for solution B, where solution C is entirely novel, so we drop solution C from consideration. Solution B remains as our only recommendation.

10.8 Recommendations

Reviving Tristan Brindle's [P0665R1] proposal, allowing for inline specializations, seems to be the best solution. The Committee should, at least, investigate what happened to the proposal after Rapperswil in 2018.

11 Paper 2: Reducing UB In Expressions Having Side Effects

11.1 Brief Historical Recap

The original 1989 C Standard that was the foundation of the C++ Standard specified that multiple modifications to the same object without an intervening sequence point is undefined behavior (UB). The C++11 Standard specified a memory model for concurrent and parallel computation that captured the trigger for undefined behavior as multiple side effects within the same value computation.

While it is not possible to solve the general problem of determining whether two value computations modify the same address, due to the presence of pointers and references within the language, a clear subset of such value computations having undefined behavior could be diagnosed, reducing the occurrence of user errors. Currently, all such diagnostics are entirely a matter of quality of implementation (QoI).

11.2 Problem Statement

Unsequenced operations on the same memory location are UB. C++ has a bad reputation for users tripping over easily avoided UB, yet common examples of unsequenced operations on the same variable are easily diagnosable.

11.2.1 Illustrative Example 1: Expressions Referencing the Same Object and Memory Location

According to the evaluation order rules, expressions operating on references pointing to the same object may lead to different results or UB with very little difference from the valid expressions:

```
void g(int i, int j) {

}

void f(int i) {
  i = i++ + 1;        // OK: i is incremented.
                      // value evaluation i
                      // side effect i++
                      // Sequence point is an assignment.

  i = i++ + i;        // UB

  g(i++, i++);        // OK: indeterminately sequenced

  int j = 1;
  g(j, i++);          // OK: i incremented (indeterminately sequenced)


  int& k = i;
  g(k++, i++);        // OK: indeterminately sequenced
}

int p(int& i, int& j) {
   return i++ + j++; // UB when `i` and `j` reference the same object
                      // otherwise unsequenced; general case
}

int q(int& i, int j) {
   return i++ + j++; // OK: unsequenced but never UB as `i` can never alias `j`
}
```

11.2.2 Illustrative Example 2: Generalized UB Case

In some cases, even in the presence of UB, the program will have a defined behavior:

```
// impl.cpp
int f(int& i, int& j) {
   return i++ + j++;      // unsequenced
                          // UB when i and j reference the same object
}

// client.cpp
extern int f(int&, int&);

int i;
int j;

f(i, j);                   // OK
f(i, i);                   // UB
```

This scenario shows the general case when the compiler does indeed require some additional information to separate valid invocations from those leading to UB. Solving the general case is outside the scope of this paper and will potentially require language extensions.

11.2.3 Business Justification

UB is the source of many difficult-to-diagnose bugs. Tracking down and fixing those bugs can be particularly difficult and expensive when those bugs look just like regular code. In particular, such subtle behavior can be trying for developers as they are learning the language and building up their skills and experience.

11.2.4 Measure of Success

Fewer bugs will be reported at all stages of the development process; in particular, finding and correcting such bugs will be accomplished before products are released to customers.

Fewer reports of the infamous difficulty of C++ relating to this issue will appear on popular discussion fora.

11.3 Probative Questions

11.3.1 How does the Standard describe the problem?

The current C++ Standard defines the following orders of evaluation.

> 8 Sequenced before_ 6.9.1 [intro.execution] is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (6.9.2 [intro.multithread]), which induces a partial order among those evaluations. Given any two evaluations *A* and *B*, if *A* is sequenced before *B* (or, equivalently, *B* is sequenced after *A*), then the execution of *A* shall precede the execution of *B*. If *A* is not sequenced before *B* and *B* is not sequenced before *A*, then *A* and *B* are *unsequenced*.
>
> [*Note 3*: The execution of unsequenced evaluations can overlap. — *end note*]
>
> Evaluations *A* and *B* are *indeterminately sequenced* when either *A* is sequenced before *B* or *B* is sequenced before *A* but which order is unspecified.

Unsequenced evaluation of the same memory location may lead to UB. If a side effect on a memory location (6.7.1 [intro.memory]) is unsequenced relative to either another side effect on the same memory location or a value computation using the value of any object in the same memory location and the two are not potentially concurrent (6.9.2 [intro.multithread]), the *behavior is undefined*.

Fully defining the order of expression evaluation will likely have runtime performance implications. We would still like to exercise the possibility of either deprecating a subset of expressions that lead to UB, defining the order of some subset of expressions, and/or providing a mechanism to declare the ordering property of an expression to assist the compiler.

What kind of code the well-defined subset of behavior is intended to support is unclear as is whether such limited support outweighs the potentially significant risk of introducing undefined behavior about which the compiler cannot warn without raising false positives in the well-defined case.

11.3.2 What are the typical consequences for unspecified evaluation order?

Unsequenced and/or indeterminately sequenced evaluation order can manifest a significant change of the program behavior between different compilers, build modes, and optimization levels. A programmer is required to know evaluation rules to avoid common pitfalls when constructing valid expressions. In cases in which the evaluation order is important, the programmer can rearrange the program and introduce well-defined sequencing points.

11.3.3 Do compilers diagnose problems today?

Currently developers are neither protected nor warned about expressions leading to UB without enabling specific warnings. Most compilers can indeed generate the warning when an expression has multiple evaluations performed with references to the same object.

— GCC warning: `-Wsequence-point`
— Clang warning: `-Wunsequenced`

11.4 Solutions

11.4.1 Proposed Solution: Deprecate Detectable Unsequenced Evaluation of the Same Memory Location Leading to UB

Deprecate unsequenced evaluation of operations on the references that can only reference the same object. Note that this solution can be applied only for cases in which the compiler can reason about the object to which the reference in question points:

```
int i = 0;
int j = i++ + i;     // i uniquely identifies the very same variable; deprecate.

int& k = i;
j = k++ + i;         // k and i reference the very same object in the current context;
                     // deprecate.

int g(int& i, int& j) {
   return i++ + j++; // i and j may or may not reference the same object;
                     // potential UB.
}
```

Most modern compilers can already warn about such use cases (but are currently subject to QoI).

- Motivating Principles
  - Make writing correct programs easier.
    - Users write incorrect code because they do not understand all the complexity of the full C++ Standards. The compiler and Standard should assist the user in writing correct programs.
- Concerns
  - Not all cases leading to UB can be recognized by compilers. A general purpose solution would require either extending the language itself or analyzing the full program to validate all references in all expressions.

11.4.2 Alternative Solution: Make Such Code Ill Formed

For the case identified above, make such code ill formed.

- Motivating Principles
  - Compilers should diagnose erroneous code.
    - Compilers should warn the user about potential errors in the code if they can.
  - Catching bugs early — at compile time — is better than later — at run time, in testing, or by customers.
    - Bugs that are caught at compile time can never cause a program to misbehave; bugs that are caught at run time are unlikely to be caught before testing and risk making it all the way to being identified by customers, who may then be suffering data corruption or worse.[6]
- Concerns
  - Inexperienced developers might believe the compiler catches more mistakes than it actually can and thus trust code with undiagnosed UB.

[6]https://dl.acm.org/doi/pdf/10.1145/251880.251992

11.4.3 Alternative Solution: Define Evaluation Order for Unsequenced Evaluations

Alternatively, we can define the order of evaluation for the expressions having references to the same object. The value evaluation and side effects associated with subexpressions should be sequenced in left-to-right order and should follow existing rules for expression evaluations (see 6.9.1 [intro.execution]p8–11):

```
// assuming i == 0 at the beginning of every example

i = i++ + i;      // value i++(0) + i(0); side-effect i++(1); assignment i = 0 + 0 = 0
i = ++i + i;      // side effect ++i(1); value ++i(1) + i(1); assignement i = 1 + 1 = 2

j = i++ + i;      // j = 0; i = 1
j = ++i + i;      // j = 2; i = 1

j = i++ + ++i;    // i++(0) + ++i(1), i = 2; j= 1
j = ++i + i++;    // ++i(1) + i++(1), i = 2; j= 2

foo(i++, ++i);    // foo(0,1); i = 2

foo(++i, i++);    // foo(1,1); i = 2
```

Note that this solution may lead to a situation in which the evaluation of expression will contextually depend on the program code and may produce different results depending of this context:

```
int foo(int& i, int& j) {
  return i++ + j;      // unsequenced evaluation
}

int i = 0;
int j = foo (i, i);    // indeterminately sequenced evaluation

int k = i++ + i;       // sequenced evaluation (if this solution is applied)
```

In general, this solution will lead to a silent change of behavior for currently well-formed programs. Also note that by defining a strict order of evaluation, we will limit the compiler's ability to generate efficient machine code.

- Motivating Principles
  - Do not cause silent breaking changes.
    - Changing the behavior of conforming code without providing diagnostics can result in production problems that are difficult to diagnose.
  - Minimize changing essential behavior in existing code with a new Standard.
    - To the extent this principle is violated, clients who previously had correct, working code could find that their programs no longer behave the same way. Such runtime incompatibility serves as a strong deterrent to upgrading to a new Standard, which can in turn force clients to dead-end support of the code at a particular Standard.
- Concerns
  - Defining order of evaluation will have a broad effect on currently existing code and may yield a different expression evaluation result.
  - A defined evaluation order will lead to a program performance degradation due to added sequence points.

11.5 Curated, Refined, Characterized, and Ranked Principles

First, we collect all the motivating principles from the solutions above and copy them verbatim into the table below. We refine the wording to ensure that each stated principle is clear and unambiguous. At this point, we might add to the bottom of the table some more general principles that are known to apply to many problems.

Next, we score the principles on importance and objectivity and use the results to rank them. We can then proceed to scoring the solutions against the principles, using a measurement scheme in which higher is better, @ is True (100%), 9 is 90%, ..., 1 is 10%, and - is False (0%).

Table 27: Importance and Objectivity of Principles

Rank	i	o	Principle ID	Principle Statement
4	9	5	CorrectUsage	Make writing correct programs easier.
5	9	5	Diagnose	Compilers should diagnose potential errors.
6	7	7	CompileTime	Catch bugs early, preferably at compile time.
1	@	@	NoSilent	Do not cause silent breaking changes.
2	@	@	ScopeImpact	Minimize changing essential behavior in existing code with a new Standard.
3	9	@	AdditionalInfo	Adding information does not change well-defined behavior to ill-formed or to different well-defined behavior.
7	5	@	Deprecated1st	Avoid making well-formed code ill formed in one Standard cycle.

11.6 The Compliance Table

Now we will score the solutions against the ranked principles. We introduce the status-quo solution as the current baseline that viable solutions must beat. The scale is @ means True (100%), 9 means 90%, ..., 1 means 10%, and - means False (0%).

Key to solutions:

A. Status Quo
B. Deprecate
C. Ill formed
D. Define Order

Table 28: Compliance Table

Rank	i	o	Principle ID	A	B	C	D
1	@	@	NoSilent	@	@	@	5
2	@	@	ScopeImpact	@	@	@	1
3	9	@	AdditionalInfo	@	@	@	1
4	9	5	CorrectUsage	5	9	9	@
5	9	5	Diagnose	5	9	9	5
6	7	7	CompileTime	-	5	7	9
7	5	@	Deprecate1st	1	@	9	1

11.7 Analysis of the Compliance Table

Row 1 — A B C D — NoSilent(@,@)

Solution D is eliminated immediately because it defines the order of evaluation that affects all previously unsequenced and indeterminately sequenced expressions.

Rows 2–3 — A B C _ — ScopeImpact(@,@) and AdditionalInfo(9,@)

Those principles do not discriminate among any of the remaining solutions because none of the solutions changes current behavior. Also note that these two rows cement the elimination of solution D due to its wide impact on already existing code.

Rows 4–5 — A B C _ — CorrectUsage(9,5) and Diagnose (9,5)

These two rows show that any diagnostics emitted by the compiler will help programmers to write correct code and avoid common errors. These two rows eliminate the status quo solution since it does not enforce any diagnostics.

Row 6 — _ B C _ — CompileTime (7,7)

Assuming compilers will warn on deprecated behavior, then both solutions B and C will diagnose the same conditions, identifying potential bugs at compile time. Solution C satisfies this principle slightly better by making the program ill formed, but neither is a complete solution as there remain many potentially unsequenced evaluations that will not be diagnosed.

Row 7 — _ B C _ — Deprecate1st (5,@)

The last principle, allowing programmers to address the warnings detected (and deprecated) by compilers before making such expressions ill formed, indicates a slight preference for solution B.

11.8 Recommendations

Looking at the scored solutions, solutions B and C are clear winners, rank closely together, and deserve further consideration. In fact, the two solutions are effectively the same solution, and we must simply choose between deprecation or immediately making the diagnosable cases ill formed, which is a decision better handled by finding consensus within the larger group.

12 Paper 3: Addressing the Most Vexing Parse

12.1 Brief Historical Recap

One of the most infamous gotchas of C++ is the most vexing parse (MVP), discussed in *Effective C++* by Scott Meyers [meyers05]. The intention is to declare a variable with a constructor call, but the declaration is ambiguous as to whether it is a variable or a function declaration; the compiler interprets it as a function declaration though that usually isn't the coder's intent. This erroneous interpretation can lead to confusing syntax error messages when the program attempts to access the intended variable and the name of the declared object is interpreted as a function pointer rather than as a variable.

Even if the intended variable is previously declared correctly without initialization (i.e., via an `extern` declaration or by being declared as a static variable in the class), when the variable is initialized with an expression that may be parsed as a function declaration, the compiler will complain about the identifier being declared inconsistently, and the program is ill formed.

12.2 Problem Statement

In multiple places in the C++ grammar, a declaration could have multiple meanings. Several of these ambiguities are resolved in surprising and nonintuitive ways, which we call *vexing parses*. We would like to reduce the number of parses in C++ that are surprising, especially for students learning the language.

12.2.1 Illustrative Example

Consider two renderings of the `make` function in the following example.

Vexing

```
template <class T>
T make() {
   T object(T());
   return object;
}

int main() {
   return make<int>();
}
```

Clear

```
template <class T>
T make() {
   extern T object(T(*)());
   return object;
}

int main() {
   return make<int>();
}
```

The vexing version looks like it *might* be trying to declare a local variable name `object` and to initialize that variable with a value-initialized temporary object of type `T`. In fact, versions of this code have exactly the same meaning: declaring a function named `object` that has the same parameter list and linkage as that original example and that must be defined in the same namespace as the `make` function template.

Clarity is achieved in this case by explicit use of the `extern` keyword to declare the function (i.e., it is not permitted to initialize an `extern` variable declaration), making the function-pointer parameter type clear rather than relying on the implicit function-to-pointer decay.

12.2.2 Business Justification

The existence of the most vexing parse issue harms the reputation of C++ and contributes to the perception that C++ is a user-hostile language full of subtleties and surprises for simple-looking code.

Vexing parses also lead to code that is subtle and easily misunderstood by human developers. The disambiguation rules for vexing parses typically produce an interpretation that is not useful since locally scoped function declarations are *significantly* rarer in practice than local variable declarations; where the human developer is writing such code, the names used will typically further mislead human readers, who expect an object rather than a function.

12.2.3 Measure of Success

Most vexing parses will occur less frequently in code.

12.3 Probative Questions

12.3.1 Does this potential ambiguity exist for all initializations?

Nonstatic data members are initialized unambiguously in constructor member initializer lists or in the class definition itself and are thus not vulnerable to the most vexing parse. However, the problem remains for block-scope variables, file-scope static variables, and namespace-scope variables.

12.3.2 Has there been any previous work to address this problem?

C++11 introduced the ability to replace the use of parentheses for initialization with the use of curly braces instead, and this syntax is unambiguously *not* a function declaration; a programmer who always uses brace-initialization will completely avoid the most vexing parse. However, brace-initialization is not entirely equivalent to the parenthetical form since the brace-list will be treated as an `initializer_list` where that interpretation is valid, and the compiler also enforces a new rule that prohibits narrowing conversions.

12.3.3 Do compilers diagnose this problem today?

Clang has a `-Wvexing-parse` option (which GNU GCC seems to ignore) that will flag some but not all instances of the ambiguous declarations but only when they're at block-scope and only when parentheses are used in the expressions declaring the type of at least one of the arguments. Note that a most vexing parse occurring at file or namespace scope will not be caught. Even if the most vexing parse warning does not occur, the error message Clang++ gives is good, describing the exact type of the function pointer referenced, and thus the coder can easily realize they've stumbled into a most vexing parse if they are aware of the type of problem. In the unlikely event that the coder really wanted a function declaration, the warning can be silenced with some typedefs to eliminate the parentheses in the arguments of the declaration.

12.4 Solutions

12.4.1 Proposed Solution: Deprecate Block-Scope Function Declarations

Deprecate function declarations that occur at block scope and would be ambiguous with a variable definition (invoking a constructor) absent a vexing parse tie-breaker rule. Such declarations would be expected to produce a deprecation warning and can be resolved (as function declarations) through use of the `extern` keyword.

- — Motivating Principles
  - — Since these would be warnings (not errors), if the intent really was for the declaration to be a function, the compile will still succeed.
  - — The coder would be warned that they have created a most vexing parse so that syntax errors they get when they try to use the identifier as a variable rather than a function will make more intuitive sense to them.
- — Concerns
  - — We have no good reason to declare a function at block-scope. C and C++ did away with nested functions, so functions are either file-scoped global, file-scoped static, namespace scoped, or methods in classes, and just declaring them directly at the appropriate scope makes sense.
  - — Note that this solution does not rule out `using` declarations of functions at block-scope, in which case the most vexing parse is not a problem. This solution would not eliminate the occurrence of the most vexing parse when declaring variables at file or namespace scope.
  - — Some eccentric programmers might have been declaring lots of functions at block-scope, and bringing their programs into compliance will be difficult, but this situation will be rare.
  - — This solution would not really be a big change because many compilers already give warnings about any MVP at block scope, and deprecation would just be a heads-up that we intend to eventually make such code ill formed.

12.4.2 Alternative Solution: Ambiguous Block-Scope Function Declarations Are Ill Formed

Functions declarations — at block scope — that would be ambiguous with a variable definition should retain that ambiguity and be ill-formed, so that developers provide a clear declaration or definition that is unambiguous to human readers of the code.

Note that function declarations can be made unambiguous by including the (implied) `extern` keyword, and variable definitions can be made unambiguous by using a different valid initialization syntax.

- — Motivating Principles
  - — Do not resolve inherent ambiguities if that leads to confusing code.
  - — Do not lose expressibility that is present in the current language.
- — Concerns
  - — This solution would break well-formed code (that is not deprecated).

12.4.3 Alternative Solution: Ambiguous Block-Scope Function Declarations Are Variable Definitions

Functions declarations — at block scope — that would be ambiguous with a variable definition should be consistently interpreted as variable definitions since that is most consistent with user expectations of that syntax in C++.

Note that function declarations can be unambiguously declared by including the (implied) `extern` keyword.

- — Motivating Principles
  - — Specification should optimize clarity for human readers over clarity for tools.
- — Concerns
  - — This solution would silently change the meaning of well-formed code.

12.4.4 Alternative Solution: Disambiguate Definitions for Declared Objects

In the case in which a variable is declared unambiguously without initialization as `extern` or as a static variable in a class and then later initialized with ambiguous syntax that could be interpreted as a function definition, currently interpreted as multiple conflicting definitions and ill formed, have the compiler defer to the original definition of the variable and interpret the latter definition as an initialization.

This solution would not affect any currently well-formed code; it would make well formed any code that is currently ill formed. In nearly all cases, the semantics would be what the programmer desired.

Current Behavior

```cpp
// a.h

struct A {
    ...data...

    A(int);
};

struct B {
    static A s_meow;
};

extern A woof;

// a.cpp
#include "a.h"

A B::s_meow(int());   // most vexing parse
                      // redefines `B::s_meow`
                      //   as a function
                      // ill formed

A woof(int());        // most vexing parse
                      // redefines `woof`
                      //   as a function
                      // ill formed
```

Proposed behavior

```cpp
// a.h

struct A {
    ...data...

    A(int);
};

struct B {
    static A s_meow;
};

extern A woof;

// a.cpp
#include "a.h"

A B::s_meow(int());   // well formed;
                      // interpreted
                      // as initialization

A woof(int());        // well formed;
                      // interpreted
                      // as initialization.
```

- Motivating Principles
  - The behavior of previously well-formed code would be unchanged.
- Concerns
  - This solution would be problematic only in cases in which someone makes a mistake and forward-declares a function whose name, unknown to them, is already an `extern`ed variable whose type is the return type of the function they intended to declare. This situation will be rare and will nearly always result in the linker complaining about multiple definitions of a variable.

12.5 Curated, Refined, Characterized, and Ranked Principles

First, we collect all the motivating principles from the solutions above and copy them verbatim, in the order in which they appear in the paper, into the table below. We refine that wording to ensure that each stated principle is clear and unambiguous. At this point, we might add to the bottom of the table some more general principles that are known to apply to many problems.

Next, we score the principles on importance and objectivity and use the results to rank them. We can then proceed to scoring the solutions against the principles, using a measurement scheme in which higher is better, @ is True (100%), 9 is 90%, ..., 1 is 10%, and - is False (0%).

Table 31: Importance and Objectivity of Principles

Rank	i	o	Principle ID	Principle Statement
12	5	@	FalseErrors	Code in which the programmer's intention is clear, sensible, and safe should not be ill formed.
3	@	5	Comprehensible	When code is ill formed, the reason should be evident.
6	9	7	PreferAmiguity	Do not resolve inherent ambiguities if that leads to confusing code.
4	9	@	Expressibility	Do not lose expressibility that is present in the current language.
10	7	3	PreferHumans	Specification should optimize clarity for human readers over clarity for tools.
1	@	7	SameSemantics	Do not quietly change essential behavior of well-formed code.
5	9	@	Deprecate1st	Do not make well-formed nondeprecated code become ill formed.
11	5	@	StillCompiles	Do not make well-formed code become ill formed.
7	7	@	CompileSpeed	Compile speed is not significantly slowed for well-formed code.
9	7	5	PrecSol	Do not preclude possibly better future solutions.
8	7	7	ExcessWarn	Do not generate an unacceptable level of warnings.
2	@	5	IsViable	There is no reason to believe that the solution is not implementable.

12.6 The Compliance Table

Now we will score the solutions against the ranked principles. We introduce the status-quo solution as the current baseline that viable solutions must beat. The scale is @ means True (100%), 9 means 90%, ..., 1 means 10%, and - means False (0%).

NB: To avoid repeating the compliance table, we add a new solution, F, here, and we describe solution F after the table analysis.

Table 32: Key to Solutions

Col		**Solutions**
A.	Status-Quo Default	**No change to the Standard**
B.	Original Proposed	**Deprecate at block scope**
C.	Original Alternate	**Ill formed at block scope**
D.	Original Alternate	**Disambiguate as variables at block scope**
E.	Original Alternate	**Disambiguate definitions for predeclared objects**
F.	Newly Proposed	**Combination of B & E**

Table 33: Compliance Table

Rank	**i**	**o**	**Principle ID**	**A**	**B**	**C**	**D**	**E**	**F**
1	@	7	SameSemantics	@	@	@	-	@	@
2	@	5	IsViable	@	@	@	@	@	@
3	@	5	Comprehensible	3	5	@	5	3	5
4	9	@	Expressibility	@	@	@	@	@	@
5	9	@	Deprecate1st	@	@	_	7	@	@
6	9	7	PreferAmiguity	_	5	@	_	1	3
7	7	@	CompileSpeed	@	@	@	@	@	@
8	7	7	ExcessWarn	@	9	@	@	@	9
9	7	5	PrecSol	@	@	9	_	@	@
10	7	3	PreferHumans	_	5	@	9	@	9
11	5	@	StillCompiles	@	@	_	@	@	@
12	5	@	FalseErrors	5	5	9	7	7	7

12.7 Analysis of the Compliance Table

Row 1 — A B C D E F — SameSemantics(@,@)

Clearly solution D changes one kind of well-defined behavior for another and thus falls out of consideration. All the other solutions satisfy this principle.

Row 2 — A B C _ E F — IsViable(@,5)

We know of no barriers to implementation for any of the remaining solutions, so all satisfy this imperative principle.

Row 3 — A B C _ E F — Comprehensible(@,5)

Errors reported due to most vexing parses can often be incomprehensible and arise from use of the surprising parse of that declaration in other statements. Compilers are free to report their own QoI diagnostics, and deprecation in the Standard might make such diagnostics more consistent but would still not require them. Hence, only solution C, which makes more parses directly into diagnosable errors satisfies this principle cleanly. While all solutions allow for QoI diagnostics, we deem those that deprecate to better satisfy this principle; hence, only solutions A and E are demoted to lowercase.

Row 4 — a B C _ e F — Expressibility(9,@)

All remaining proposed solutions retain the ability to express the same set of valid C++ constructions, even if some require existing code to change to express that exact same intent. Therefore, all proposed solutions satisfy this principle.

Row 5 — a B C _ e F — Deprecate1st(9,@)

Solution C is immediately rejected for breaking well-formed code that has not been deprecated; it might become viable for a future Standard but goes no further for C++26. The remaining solutions all cleanly satisfy this principle.

Row 6 — a B _ _ e F — PreferAmiguity(9,7)

This principle is somewhat subjective since it states a preference rather than a strict rule. Solutions A and E, however, clearly resolve ambiguities in the grammar, and they fall foul of this principle; one could argue that the resolution of solution E is motivated sufficiently to pass the "prefer" barrier, so it gets a nonzero rating, but it is still excluded from consideration for this revision of the paper. Solution F has the same problems as solution E but also some of the advantages of solution B, so we split the difference and demote solution F while retaining it for consideration of the remaining principles.

Rows 7–10 — _ B _ _ _ f

For ease of reading, the scores for solutions B and F against the principles that scored 7 for importance are repeated here.

Rank	**i**	**o**	**Principle ID**	**B**	**f**
7	7	@	CompileSpeed	@	@
8	7	7	ExcessWarn	9	9
9	7	5	PrecSol	@	@
10	7	3	PreferHumans	5	9

Scanning this table of the principles that scored 7 on importance, we can see that compliance is close for all the remaining candidate solutions until row ten, where the status-quo solution is eliminated. Solution F has a slightly better score than solution B, but since we are now considering lower-priority principles and the difference is not disqualifying, we retain the overall preference for solution B over solution F.

Rows 11–12 — _ B _ _ _ f — StillCompiles(5,@) and FalseErrors(5,@)

For ease of reading, the scores for solutions B and F against the remaining principles are repeated here.

Rank	**i**	**o**	**Principle ID**	**B**	**f**
11	5	@	StillCompiles	@	@
12	5	@	FalseErrors	5	7

Again, comparing the last two principles, we see a slight preference for solution F over solution B in the lower-priority principles, suggesting either might be a reasonable choice.

12.8 Newly Proposed Solution: Deprecate at Block Scope and Disambiguate

After preliminary analysis, we saw that our original proposed solution B could benefit from our alternate solution E. Hence, we created a new solution, F, that combines the benefits of both. That is, we propose to deprecate function declarations that occur at block scope that would be ambiguous with a variable definition (invoking a constructor) absent a vexing parse tie-breaker rule. Additionally, in the case in which a variable is declared unambiguously without initialization as `extern` or as a static variable in a class and then later initialized with ambiguous syntax that could be interpreted as a function definition, currently interpreted as multiple conflicting definitions and ill formed, the compiler can defer to the original definition of the variable and interpret the latter definition as an initialization.

- Motivating Principles
  - Since these notifications would be warnings (not errors), if the intent really was for the declaration to be a function, the compile will still succeed.
  - The coder would be warned that they have created a most vexing parse so that syntax errors they get when they try to use the identifier as a variable rather than a function will make more intuitive sense to them.
  - The behavior of previously well-formed code would be unchanged.
- Concerns
  - Some eccentric programmers might have been declaring lots of functions at block-scope; bringing their programs into compliance will be difficult, but this situation will be rare.
  - This solution would be problematic only in cases in which someone makes a mistake and forward-declares a function whose name, unknown to them, is already an `extern`ed variable whose type is the return type of the function they intended to declare. This situation will be rare and will nearly always result in the linker complaining about multiple definitions of a variable.

12.9 Recommendations

After reviewing all solutions against all principles, we come to the conclusion that two solutions are preferred above all others. Solution B, "Proposed Solution: Deprecate Block-Scope Function Declarations," offers the preferred direction if we lean in favor of never disambiguating the grammar, forcing developers to write explicit code. Solution F, "Deprecate at Block Scope and Disambiguate," which combines solution B with solution E, "Alternative Solution: Disambiguate Definitions for Declared Objects," leans in favor of allowing disambiguation in favor of a human reader where we believe that disambiguation is sufficiently obvious.

13 Paper 4: Deprecate Delete of a Pointer to an Incomplete Type

This example matured significantly during the preparation of this paper and is submitted as a formal proposal as paper [P3144R0].

14 Acknowledgments

Our appreciation for the inspiration for this paper goes to JF Bastien, Thomas Köppe, Jake Fevold, and Joshua Berne for their recent applications of those principles. We also owe thanks to John Spicer and Timur Doumler, who chair the SG21 working group where much of this work has been put to good use in the development of the Contracts MVP for C++26.

Our gratitude goes to Bjarne Stroustrup for documenting the principles for the design of the language in [stroustrup94] and for consistently championing their use.

We offer our thanks to this paper's team leader, Nathan Roy, and to our survey takers, Brian Bi, Steven Breitstein, and Henry Mike Verschell.

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

15 References

[lakos22] John Lakos, Vittorio Romeo, Rostislav Khlebnikov, and Alisdair Meredith. 2022. Embracing Modern C++ Safely.

[meyers05] Scott Meyers. 2005. Effective C++.

[N2660] Lawrence Crowl. 2008-06-13. Dynamic Initialization and Destruction with Concurrency.
https://wg21.link/n2660

[N4230] R. Kawulak, A. Tomazos. 2014-10-10. Nested namespace definition (revision 2).
https://wg21.link/n4230

[P0665R1] Tristan Brindle. 2018-05-06. Allowing Class Template Specializations in Associated Namespaces (revision 1).
https://wg21.link/p0665r1

[P2723R0] JF Bastien. 2022-11-16. Zero-initialize objects of automatic storage duration.
https://wg21.link/p2723r0

[P2723R1] JF Bastien. 2023-01-15. Zero-initialize objects of automatic storage duration.
https://wg21.link/p2723r1

[P2754R0] Jake Fevold. 2023-01-24. Deconstructing Avoiding Uninitialized Reads of Auto Variables.
https://wg21.link/p2754r0

[P2795R0] Thomas Köppe. 2023-06-13. Correct and incorrect code, and &quot;erroneous behaviour&quot;
https://wg21.link/p2795r0

[P2834R1] Joshua Berne, John Lakos. 2023-06-08. Semantic Stability Across Contract-Checking Build Modes.
https://wg21.link/p2834r1

[P2932R0] Joshua Berne. 2023-09-13. A Principled Approach to Open Design Questions for Contracts.
https://wg21.link/p2932r0

[P3113R0] Timur Doumler. Contract Assertions, the `noexcept` Operator, and Deduced Exception Specifications.

https://wg21.link/p3004r0

[P3144R0] Alisdair Meredith, John Lakos, et al. Deprecate Delete of a Pointer to an Incomplete Type.
https://wg21.link/p3144r0

[stroustrup94] Bjarne Stroustrup. 1994. The Design and Evolution of C++.