Design Considerations for Class Invariants

Document #:	P4262R0
Date:	2026-07-15
Project:	Programming Language C++
Audience:	EWG

Reply-to: Joshua Berne <jberne4@bloomberg.net>

Abstract

Class invariants are a natural extension of the C++26 Contracts facility, but specifying them well requires answering fundamental design questions about when invariants should hold, how checking interacts with access control, and how to avoid prohibitive runtime overhead. Every language that has attempted class invariants has encountered the same set of challenges — and no two have arrived at the same answers. This paper enumerates the open questions that must be resolved before a class invariants proposal for C++ can be made, explores the design space, and identifies possible directions forward.

Contents

Revision History

Revision 0

- Original version of the paper

1 Introduction

A fundamental aspect of type design is identifying the invariants that a class will maintain. When we say that a `std::vector` always has `size() <= capacity()`, or that a `std::string`'s internal buffer is always null-terminated, we are describing class invariants — conditions that hold whenever an object is in a state where external code might observe it. The ability to *check* these invariants at runtime is a powerful tool for identifying bugs, and class invariants are one of the most commonly requested extensions to the C++26 Contracts facility.

We have been investigating the design space for class invariants in C++ for several years, including an initial sketch of a possible design in [P2755R1]. The more we research this problem — including the experience of other languages and the constraints imposed by C++'s compilation model — the clearer it becomes that we do not yet have a fully satisfying solution.

At first glance, many people jump to what seems like an easy and useful solution — declare a predicate in a class, check it on entry and exit of public member functions. Every language that has tried this, however, has discovered that the details are far more complex than they appear. The questions of *when* an invariant should hold, *which* function boundaries trigger checking, and *how much* overhead is acceptable have led to fundamentally different answers across Eiffel, D, Spec#, Ada, and others.

One major source of that difficulty is that two distinct questions are often conflated:

1. When do we *expect* the invariants to be true? That is, at what points in a program's execution should an object's invariants hold as a matter of correctness?
2. When is it *practical* to evaluate those expectations with a checked semantic? That is, given the cost of evaluation, at which of those points should we actually perform runtime checks?

The first question is fundamentally about the semantics of the language feature — it defines what it *means* for a program to have correct invariants. Our answer to this question must provide a guarantee that is both useful to reason about — one that results from checking at all of these points — and free of false positives for typical C++ workflows. The answer must be ergonomic while providing real utility.

The second is about controlling runtime cost, and is in large part addressable through the same mechanisms we already have for other contract assertions: evaluation semantics and labels.

Separating the questions is important, because "It would be too expensive to check that property all the time" is a good reason to default to not checking but *not* a good reason to say that the property is not expected to be true.

The first question is the primary focus of this paper, as it is where the genuinely novel design challenges lie. We will, however, also identify where the second question raises issues that go beyond

what existing contract-assertion controls provide.

2 Prior Art

Class invariants have been attempted in a number of languages, each arriving at different tradeoffs. Understanding their experience is essential before making design decisions for C++.

2.1 Eiffel

Eiffel [Meyer1997] is the origin of Design by Contract and certainly one of the earliest tools for encoding explicit class invariants. Invariants are declared in an `invariant` clause at the end of a class and are checked on entry and exit of every public *qualified* call — that is, calls of the form `x.f(...)` from outside the class.

Crucially, invariants are *not* checked on unqualified (internal) calls. This allows an object to temporarily violate its invariant during a sequence of internal operations. However, this leaves the *callback problem* unresolved: if a public method calls external code that calls back into the same object, the invariant may not hold at the point of re-entry, and Eiffel does not detect this.

Additionally, Eiffel only checks invariants on the *target* of a qualified call — not on other arguments of the same type. If `x.f(y)` is called where both `x` and `y` are of the same class, only `x`'s invariant is checked, even if `f` modifies `y`'s state. This means that operations analogous to C++ move assignment, where the source object should have its invariants verified after being modified, are not covered.

2.2 D

The D language [Dlang] checks class invariants on entry and exit of every `public` or `protected` non-static member function. This checking is unconditional — if a public method calls another public method on the same object, the invariant is checked again at the nested call boundary. This means that D will detect reentrancy bugs, but it also means that any public method that temporarily breaks the invariant cannot delegate to another public method, even on the same object. Unconditional checking is the brute-force approach whose overhead Midori found unacceptable.

Older D community documentation sometimes describes a per-object flag that would suppress invariant checking during nested public calls on the same object. Such a flag does not appear in the current D specification, and empirical testing with the reference compiler (DMD 2.112) confirms that no such suppression occurs. The limitations of D's current approach are an active topic of discussion within the D community [DlangInvariantIssue].

To avoid infinite recursion, D prohibits calling public or exported member functions from within an invariant expression itself. This restriction means that invariants in D cannot be expressed in terms of the class's public interface and must instead directly reference internal state — a significant limitation on how invariants can be written.

As with Eiffel, D only checks invariants on the `this` object — not on other arguments of the same type passed to a member function. A function that modifies a passed-in argument's state will not trigger invariant checking on that argument when it returns.

Furthermore, because D's invariant checks are tied exclusively to public method entry and exit, any modification of an object's state that bypasses a method call (e.g., directly writing to a public field, or modifying through a reference obtained from a container) will not trigger an invariant check at all. The violation is only detected later, when the next public method is called on that object — which might be far removed from the point where the invariant was actually broken.

2.3 Spec# and Code Contracts

Microsoft Research's Spec# [Leino2004] introduced the most rigorous treatment of object invariants. Rather than checking on every method boundary, Spec# requires an explicit `expose` block to put an object into a "mutable" state where its invariant need not hold. Outside of `expose` blocks, the object is "valid" and its invariant is guaranteed.

This *ownership-based methodology* solves the callback problem (you cannot re-enter a valid object while it is exposed) but at the cost of significant annotation burden. The approach was ultimately considered too heavy for mainstream use and Code Contracts for .NET was discontinued.

2.4 Ada 2012

Ada's type invariants [Ada2012RM] take an approach most closely aligned with what we will describe as the "bubble" concept. Invariants are checked when an object crosses the *package boundary* — the boundary between the package where the private type is declared and external code. Internal operations within the package can freely violate the invariant temporarily.

This design has relatively low overhead (checks occur only at module interfaces, not every call) and maps naturally to Ada's strong encapsulation model.

Notably, Ada checks invariants on *all* parameters of the private type that cross the package boundary — not just the primary "self" parameter. If a subprogram takes two `in out` parameters of the invariant type, both are checked on return. Furthermore, this checking extends to elements within composite types: if an array of invariant-type objects is passed across the boundary, the invariants of modified elements are checked. This comprehensive approach means that Ada would catch the equivalent of a C++ move operation leaving the moved-from object in an invalid state.

2.5 Midori

Microsoft's Midori OS project initially adopted the Eiffel approach of checking invariants on every public method boundary. As documented by Duffy [Duffy2016], the overhead was unacceptable for systems-level code. They evolved toward selective checking at "publication points" where objects become visible to other components, combined with heavy use of static verification.

2.6 Dafny

Dafny [Leino2010] takes the most radical approach: invariants are purely static proof obligations verified at compile time. There is no runtime checking at all. Each method must explicitly state `requires Valid()` and `ensures Valid()`. This eliminates runtime cost entirely but places the full burden of specifying checking points on the programmer.

2.7 Lessons for C++

Every system that started with "check the invariant on every public method entry and exit" either:

- encountered the callback/reentrancy problem and had to add exceptions to the rule (Eiffel, D),
- found the performance overhead unacceptable for systems code (Midori), or
- moved toward boundary-based or ownership-based semantics that give the programmer control over when the invariant is expected to hold (Spec#, Ada).

A second cross-cutting distinction emerges around *which objects* are checked at a given boundary:

- Eiffel and D check only the receiver object (`this` / the target of the qualified call). Other arguments of the same type that are modified during the call are not checked, even though they have also crossed the encapsulation boundary.
- Ada checks *all* objects of the private type that cross the package boundary, including non-self parameters and elements within composite types like arrays.
- Dafny makes all such decisions explicit — the programmer must name every object whose invariant should hold.

For C++, where move operations routinely modify a non-`this` argument and where friend functions can operate on multiple objects simultaneously, the Ada approach of checking all objects that cross the boundary has significant appeal. However, Ada's module system is much simpler than C++'s combination of access control, friendship, and inheritance, and it is not yet clear whether Ada's approach can be faithfully mapped onto C++'s model.

The lesson for C++ is that we must identify a principled rule for *when* invariants should hold that avoids both the unsoundness of Eiffel's approach and the impracticality of brute-force checking, while remaining expressible within C++'s compilation and access-control model. Whether that rule is closer to Ada's boundary-based approach, to a fine-grained approach that checks at all function boundaries (Section 3.5.13), or to some hybrid remains an open question.

Within WG21, we previously published [P2755R1] which explored some aspects of a potential solution for invariants (in Sections 2.2.11-2.2.13). Based on the variety of invariant-related use cases raised in [P1995R1], that design focused on making sure we could identify invariants that impacted the different layers of access control provided by C++ — `public`, `protected`, and `private`. It also identified the requirements that we are likely to want to handle `const` and non-`const` member functions differently, as well as needing specific treatment to handle cases where there are `mutable` members. That exploration showed that we need a rich and flexible feature, but was far from exhaustive. (The exploration in this paper has evolved from it, and it is clear that we still need to find the right level of complexity and usability as we continue exploring the design space).

Another paper, [P3361R1], explored class invariants from a philosophical perspective. That paper frames invariants as constraints on data members and as preconditions on destructors — if all methods reestablish the invariant, the destructor can assume it holds. [P3361R1] also notes that move operations must not break the invariant of the source object (since it must still be destructible), and raises the practical question of how invariant checking interacts with mutex locking in concurrent

code. However, [P3361R1] assumes that friends "behave nicely" and should not break invariants, removing any responsibility for the language feature to provide a way for the user to choose to check such assumptions — a design choice that we have not made elsewhere in the general evolution of Contracts in C++.

3 Feature Design

In this section we describe the aspects of the feature that need to be designed. We begin by establishing terminology, introducing a running example, and then working through the design questions that the example reveals.

3.1 Definitions

Before proceeding, we establish terminology that will be used throughout this paper:

- A *class invariant* (or simply *invariant*) is a property that is expected to hold for every object of a class at every point in the object's lifetime when the object is passed between different regions of code.

  In general invariants benefit from *encapsulation* by putting constraints on properties that can be changed only by functions with privileged access to the object's internal state, such as members and friends. Structs and types with public members, however, rely on any code that uses them working together to maintain the object's invariants.

- A *usual assertion* is a property that we typically expect to hold for objects of a class but that may not hold in all valid states — for example, a moved-from object might satisfy its invariants (it is still destructible and assignable) but not its usual assertions (it may not be usable for its intended purpose).

  Many users think of usual assertions when they talk about invariants, and it is often easy to conflate the two. Long WG21 discussions about whether a moved-from object must satisfy its class invariants stem from exactly this distinction — such objects absolutely must satisfy their class invariants, but can certainly be in a state where they satisfy none of the other usual assertions.

- A *bubble* is the region of execution within which an object's invariants may temporarily be violated. Code inside the bubble — member functions, friends, and potentially derived classes accessing protected members — can break and restore invariants as part of implementing the class's operations.

  A design for invariants can involve very small bubbles (every single function) or very large bubbles (anything flowing from a call to a public member function).

  Bubbles like this have also been referred to as "invariant critical sections", though that description is most apt when a bubble is entirely syntactic.

- The *boundaries* of the bubble are those points where an object transitions between the inside and outside of the bubble — entering privileged code from unprivileged code, or returning

from privileged code to unprivileged code. These are the points where we expect invariants to hold and where checking is appropriate.

A design for invariants can give bubbles a strict boundary — where any possible way an object goes from outside the bubble to inside is a place where we check — or a porous bubble where we identify only specific crossings that are potentially checked.

3.2 Motivation

When considering a new feature that is often requested, it can become easy to forget to think about *why* the feature might be useful due to the regularity of people asking for it on their own, for their own reasons. Having a clear focus in mind about what purpose invariants have can, however, greatly help in determining an appropriate scope for the feature. More importantly, it can be pivotal in deciding which design tradeoffs are acceptable and which ones compromise the initial goal of the feature in the first place.

We have seen a few potential motivations for invariants expressed, and each one leads to a different set of requirements for the feature:

- In their most common form in the wild, they achieve the goal of providing syntactic sugar for writing a group of assertions you expect to duplicate in many places.
- A better motivated reason with the same end result for a simple invariants feature is to improve documentation and reduce the cognitive load when using a type. While that often means a simpler feature is better, if the situations where an invariant can really be relied upon are hard to unravel, the feature will have failed to deliver true benefits.
- A more rigorous goal is to be able to identify *neighborhoods* where an invariant *could* be broken and then check the invariant on all boundaries of those neighborhoods. These neighborhoods could then be viewed as a form of *encapsulated bubble* where confidence can be gained without anything outside the bubble needing to be aware of, or responsible for the invariants.

  Sound uses of this model require that the entire boundary of a neighborhood be checkable and that the invariants themselves be written such that they cannot be broken outside of the bubble. The second property can be left to users — access control provides the tools required with acceptable tradeoffs due to the known edge cases where it can be violated (all of which are generally regarded as inappropriate to use in real software). The first property, however, is where the invariants feature itself must deliver.

Each of the potential motivating purposes needs to be considered as the feature is designed and tradeoffs are evaluated.

3.3 `ConstantSumGame` — an Example to Reference and Revisit

To make the design questions concrete, we introduce a class that will serve as a running example throughout this paper. `ConstantSumGame` models a game in which a fixed number of points are distributed among players, with the fundamental constraint that the total number of points in the system is always tracked in a `d_totalPoints` member variable — every point awarded to one player

must come from somewhere[1].

```
class ConstantSumGame {
  std::string                                        d_name;
  int                                                d_totalPoints;
  std::unordered_map<std::string, int, /* ... */> d_scores;

  int sumOfScores() const;
    // Compute the total score of all players.

public:
  invariant(sumOfScores() == d_totalPoints);

  // ...
};
```

The requirement to adjust points to always balance between players (or update the total points) is very similar to common invariant examples in other literature such as bank account transfers.

Now let's consider a few key functions in this class and how those functions interact with invariants:

- The `sumOfScores` function:

  ```
  int ConstantSumGame::sumOfScores() const {
    return std::accumulate(
      d_scores.begin(),
      d_scores.end(),
      0,
      [](auto&& acc, auto&& v) { return acc + v.second; }
    );
  }
  ```

  The first example to consider is the function we use in the invariant itself. Were it a public function, one would expect it to have a postcondition that it always returns `d_totalPoints`. As a private utility function, we might want that postcondition, but then we can never risk calling it in situations where we have adjusted the invariants.

- The move constructor:

  ```
  ConstantSumGame::ConstantSumGame(ConstantSumGame&& source)
  : d_name(std::move(source.d_name))
  , d_totalPoints(std::move(source.d_totalPoints))
  , d_scores(std::move(source.d_scores))
  {}
  ```

  The above implementation might be more than sufficient, but it leaves the source object in a state where the stated invariants might not be satisfied — if its total score was not previously `0`, its scores map is now empty (so `sumOfScores()` returns 0) while `d_totalPoints` retains its original value, and thus is invalid. We could adjust our move constructor to zero out the source

[1] We will assume that the `d_scores` member is declared so that transparent lookup with, for example `std::string_view`, is supported.

total score, but that would be wasted work for what is an otherwise unusable game object (with an unspecified name).

If we don't check the invariants on the `source` object after the move constructor we instead end up in a situation where we might have introduced a bug in this code but are failing to capture it at a useful time — some function much later will check the invariants and break because the scores on that object do not add up.

This is exactly the case where we might want to limit the scope of the checks added by the feature to be "usual assertions" instead, opting out of applying them to the source object in the move constructor, the target object in the assignment operator, and the destructor.

- The `name` accessor:

```
const std::string&  ConstantSumGame::name() const {
    return d_name;
  }
```

Here is a function that works correctly regardless of whether the invariants of the type currently hold. More notably, this is also a function we are likely to want to use when logging any information, including in those cases where we choose to log something while invariants do not hold.

- Next, consider a function that transfers points from one player to another:

```
void ConstantSumGame::transfer(
  std::string_view player1,
  std::string_view player2,
  int points)
{
  d_scores[player1] -= points;
    // sumOfScores() is now d_totalPoints - points
  d_scores[player2] += points;
    // sumOfScores() is now d_totalPoints again
}
```

Between the two lines of our function body, the invariant `sumOfScores() == d_totalPoints` is broken: we have removed `points` from one player's score without yet adding it to another, so the sum of all scores is temporarily `d_totalPoints - points`. There is no practical way to avoid this temporary breakage in a two-step operation.

Of course, that breakage impacts our lives if we attempt to add some tracing to our function body:

```
void ConstantSumGame::transfer( /*...*/ )
{
  std::print("Game[{}] Transferring {} points from {} to {}",
             name(), points, player1, player2);
  std::print("Game[{}] Subtracting {} points from {} (current points: {})",
             name(), points, player1, d_scores[player1]);
  d_scores[player1] -= points;
  std::print("Game[{}] Adding {} points to {} (current points: {})",
```

```
            name(), points, player2, d_scores[player2]);
  d_scores[player2] += points;
}
```

Surely we're logging excessively here, but in a more general case we might be inserting such tracing into a much more involved function. In this case, however, we encounter one of the fundamental problems with checking invariants on `name()` — our final `print` message above will call `name()` with invariants broken.

- Next we might have a static member function (or even a friend function), that does something even more involved with two instances of `ConstantSumGame`:

```
ConstantSumGame ConstantSumGame::merge(
  ConstantSumGame&& lhs,
  ConstantSumGame&& rhs)
{
  ConstantSumGame output(std::format("{} + {}", lhs.name(), rhs.name()));

  output.d_totalPoints = lhs.d_totalPoints + rhs.d_totalPoints;
    // now our invariants on output are broken.

  output.d_scores = std::move(lhs.d_scores);
    // now the invariants of lhs are broken.

  for (auto&& ps : rhs.d_scores) {
    output.d_scores[ps.first] += ps.second;
  }
    // finally output's invariants are restored, but
    // when do we check that?

  return output;
}
```

The consideration here is that we want to trace back any bug resulting from the breaking of the invariants of `lhs` and `rhs` to a call to `merge`. We also might want to verify that the invariants of our output object are met, as any failure in our implementation to guarantee that is certainly the fault of `merge`. In none of these cases does the bug trace back to a member function being invoked on an object of type `ConstantSumGame`.

- A more indirect case comes up when we consider a similar function (whose full implementation we won't bother showing) that merges together the contents of a collection of games:

```
ConstantSumGame ConstantSumGame::mergeMany(
  std::vector<ConstantSumGame> &&games)
{
  ConstantSumGame output;

  if (!games.empty()) {
    output = std::move(games[0]);
    for (std::size_t i = 1; i < games.size(); ++i) {
      output = merge(std::move(output), std::move(games[i]));
    }
```

```
    }

    return output;
}
```

Here, our function doesn't take as input a game by reference or value, we instead take a collection object. The game objects whose invariants we are breaking do so as the return values of `std::vector::operator[]`.

Anyone expecting to use that container of games again later will certainly want to trace bugs resulting from broken invariants in those objects back to this function. The question, of course, is where in this chain we choose to require that breakage to be tracked.

- Finally, we consider a function that delegates to unprivileged functions as part of its implementation:

```
void ConstantSumGame::zeroSum()
    // update the sum of all scores to be zero
{
    int remaining = 0;
    std::for_each(
      d_scores.begin(),
      d_scores.end(),
      [&](auto&& ps) {
        ps.second -= d_totalPoints / d_scores.size();
        remaining += ps.second;
      }
    );

    // sanity check
    contract_assert(
      static_cast<std::size_t>(std::abs(remaining)) <= d_scores.size());

    std::for_each(
      d_scores.begin(),
      d_scores.end(),
      [&](auto&& ps) {
        if (remaining > 0) { ps.second--; remaining--; }
        else if (remaining < 0) { ps.second++; remaining++; }
      }
    );

    d_totalPoints = 0;
}
```

Looking carefully at both of the above loops, we are asking `std::for_each` to invoke a function object that has an internal copy of our `this` pointer pointing to an object whose invariants are currently broken. Any rule we write that would allow this case should work equally well if we write this as a lambda, a fully featured function object, or a member function invocation using `std::bind` or some similar facility.

Certainly this code could be restructured to keep invariant breakage much more local (adjusting `d_totalPoints` each time we touch a player's score, for instance, and precomputing the total to subtract). Demanding that users do so, however, requires that users understand clearly that the boundary of their closure object is another place where we expect the invariants to be met. It's also not hard to envision cases where a more involved set of related invariants would not be as simple to establish within each individual lambda body.

- The `mergeMany` function above also illustrates a question about containers and indirect access (Section 3.5.11): when `mergeMany` receives a `std::vector<ConstantSumGame>`, should the invariants of every element in that vector be checked at the function boundary? The vector itself has its own invariants (e.g., `size() <= capacity()`), but the `ConstantSumGame` objects inside it are what we are about to modify — and `mergeMany` will leave many of them in a moved-from state when it returns.
- Consider what happens if `transfer` throws an exception between the two score adjustments — the invariant is broken, and stack unwinding will expose the broken object to any code with access to it (Section 3.5.12). Similarly, if the lambda in `zeroSum` throws partway through the first loop, the game object is left with `d_scores` partially adjusted but `d_totalPoints` unchanged. Whether the invariant should be checked during unwinding, and what to do if it fails, is a design question that interacts with both the bubble definition and exception safety guarantees.

The key design questions to think about as we proceed all lie somewhere within the above example — where should we be requiring the invariants to be met and how will that require we restructure our code to satisfy those demands. A truly useful solution will require no restructuring and be ergonomic to understand and deploy, while still finding bugs in cases where we do make clear mistakes.

3.4 Syntax Considerations

Throughout this section, all names — including `invariant`, `assert_invariants`, `no_invariant`, and `audit` — are strawman syntax used for illustration. The semantic questions are what matter at this stage; the specific spelling of any facility can be resolved later.

3.4.1 Invariant Declarations

Invariants are declared within a class body using a syntax parallel to other contract assertions in C++26, with a new introducer such as `invariant`:

```
class C {
public:
  bool isValid();

  invariant(isValid());
};
```

The introducer shown here, `invariant`, might be feasible but it is possible we will need to introduce a keyword instead of making this an identifier with special meaning, as it may be ambiguous in class scope whether the keyword is introducing an invariant declaration or a member variable declaration with gratuitous extra parentheses (or, in a class named `invariant`, a constructor). If it is ambiguous,

alternative identifiers such as `contract_invariant` will need to be considered (see [P2961R2] for some further discussion on this subject).

This syntax is a natural extension of the existing Contracts syntax. As with other contract assertions, invariant predicates should allow any C++ expression — there is no reason to impose restrictions on what can appear in an invariant that do not also apply to `pre`, `post`, or `contract_assert`.

One might consider restricting invariant expressions to avoid calling functions that would immediately recurse (e.g., public member functions whose entry triggers the invariant check that contains the call). D takes this approach, prohibiting public method calls in invariants entirely. Such a restriction, however, can make much idiomatic C++ harder to write — accessors to member data often do involved calculations over time (as classes evolve and go through various forms of refactoring), and not being able to leverage putting such logic in a single place would vastly increase the maintenance burden invariants might impose. Fixing the issue requires either manual exclusions, implicitly skipping checks on `const` member functions, or checking on boundaries that is aware of the call site coming from within an invariant check.

As with `pre`, `post`, and `contract_assert`, this syntax needs similar support for having an optional assertion-control expression (a label, as proposed in [P3400R4]). Labels become important when invariants have different evaluation costs — for example, a `ConstantSumGame` might separate its cheap invariant from an expensive one:

```
invariant(sumOfScores() == d_totalPoints);          // O(n)
invariant<audit>(                                   // O(n log n)
  allScoresWithinBounds(d_scores, d_rules));
```

We expect `invariant` to introduce a new value for `std::contracts::assertion_kind`, paralleling `pre`, `post`, and `contract_assert`.

Should we choose to distinguish invariants for different access control levels, the syntax lends itself to specifying that as all other class members do — based on where in the class definition the invariant declaration is placed. We will explore that decision further in Section 3.5.2.

Other properties, such as whether an invariant applies to all functions or just to non-`const` member functions, would require additional specifiers added on to the invariant declaration.

3.4.2 Integration with C++26 Contracts

Class invariants must integrate cleanly with the existing C++26 Contracts infrastructure. In particular, when an invariant violation is detected, it should invoke the same contract-violation handler as any other contract assertion, producing a `std::contracts::contract_violation` object with appropriate metadata.

The `assertion_kind` enumeration will need new values to distinguish the different contexts in which an invariant is evaluated. At minimum:

- `invariant_pre` — the invariant is being checked as a precondition (on entry to a function or when an object crosses into a bubble).
- `invariant_post` — the invariant is being checked as a postcondition (on exit from a function or when an object leaves a bubble).

These distinct kinds allow labels and the assertion-control framework to configure evaluation semantics differently for invariant checks at function entry versus exit. For example, a user might choose to check invariants only on exit (where a violation indicates the function broke the invariant) and not on entry (where a violation indicates a prior function failed to maintain it) in builds where cost is a concern.

If a manual checking mechanism (Section 3.4.3) is adopted, a third kind value — perhaps `invariant_manual` — would identify evaluations triggered explicitly by the programmer rather than automatically at a boundary crossing. A separate kind value allows the violation handler to distinguish between automatic and manual checks, which may be useful for diagnostic purposes.

The evaluation semantic of an invariant check (whether to *ignore*, *observe*, *enforce*, or *quick-enforce*) should be determined by the same mechanisms used for other contract assertions: implementation-defined build configuration, assertion-control objects ([P3400R4]), and the labels attached to the invariant declaration.

3.4.3 Manual Invariant Checking

The need for explicitly identifying places where invariants should be satisfied varies based on the coarseness of the bubbles we define, but never goes away. With a tight bubble that checks at many boundaries automatically, manual checking is less critical — but it remains useful for verifying that an invariant has been restored at a specific point mid-function. With a broader bubble that checks only at widely-spaced boundaries, manual checking becomes essential for any intermediate verification.

Regardless of bubble granularity, there are situations where a programmer may want to explicitly request that an object's invariants be evaluated:

- A function wants to verify that it has successfully restored an object's invariants at a specific point before continuing with further operations.
- During debugging, a developer wants to narrow down where in a sequence of operations an invariant was broken.
- The automatic boundary identification does not cover a case the programmer knows is important.

We envision a keyword and expression form that takes an object and optionally a label. For example, in a `ConstantSumGame` member function that performs a multi-step rebalancing of scores, we might want to verify that the invariant has been restored at an intermediate point before continuing:

```
void ConstantSumGame::rebalance() {
  // ... complex multi-step score adjustments ...
  assert_invariants(*this);  // verify invariant restored
  std::print("Game[{}] rebalance complete\n", name());
}
```

With an optional label to control the evaluation semantic:

```
assert_invariants<audit>(*this);
```

The design of this expression — including whether it is a keyword, a library facility, or something else — remains an open question.

Note that `assert_invariants` is the conceptual dual of the `no_invariant` opt-out described in Section 3.4.4: one expands the set of points where invariants are evaluated, while the other contracts it. Together they provide full manual control over invariant evaluation at any point where the automatic rules are not in sync with the intent of the software being written.

3.4.4 Opting Out of Invariant Checking

Regardless of how the bubble is defined, there will be situations where the automatic checking rules are too aggressive for a particular piece of code. The tighter the bubble, the more frequently this arises — but even a broad bubble may need opt-out annotations for edge cases that the automatic rules cannot anticipate.

Opt-out mechanisms allow the programmer to suppress invariant checking at specific points where they are in control of the source code, acknowledging that the object may be in a broken-invariant state and taking responsibility for restoring it before the object becomes visible to unprivileged code again.

We envision a `no_invariant` qualifier that can be applied in two contexts:

On function parameters and the receiver. A `no_invariant` qualifier on a parameter declaration suppresses invariant checking for that parameter when the function is entered or exited. Similarly, it could be applied to the implicit object parameter to suppress checking on the receiver. Applied to `ConstantSumGame`, the `merge` function and a hypothetical `unsafeReset` might look like:

```
class ConstantSumGame {
  // ...
  static ConstantSumGame merge(
    no_invariant ConstantSumGame&& lhs,
    no_invariant ConstantSumGame&& rhs);
public:
  // receiver's invariant not checked on entry/exit
  void unsafeReset() no_invariant;
};
```

In `merge`, the `no_invariant` on both parameters means that invariants are not checked on either game when the function is entered or exited. This directly addresses the multi-object friend function case: the programmer declares that these objects may be in a broken-invariant state during the function's execution.

As an operator on object references. A `no_invariant` operator applied to an expression suppresses invariant checking when an object is moved into or out of a context that would normally trigger a check. This addresses the container case:

```
void batchModify(
    std::vector<ConstantSumGame>& games) {
  for (auto& game : games) {
    auto& ref = no_invariant(game);
```

```
    // ... modify ref, invariant may be broken ...
  }
  // ... restore all invariants ...
}
```

This approach has several advantages:

- The default behavior is *safe* — invariants are checked everywhere unless explicitly suppressed.
- The points where checking is suppressed are visible in the source code, making it clear where a programmer has taken responsibility for maintaining correctness manually.
- The specification of the bubble itself can be simpler: it need not account for every edge case, because the programmer has the tools to handle those cases explicitly.

One argument in favor of this approach is that `no_invariant` annotations will always appear in code that already has privileged access to the class — member functions, friends, or derived classes — since only privileged code can break an invariant in the first place. This means the annotations are co-located with the invariant definition itself, making them feasible to require and easy to audit.

An open question for this approach is how it interacts with protected access. A derived class accessing protected members has a level of privilege that is higher than public but lower than private. If the tight bubble includes protected boundaries, derived classes might need `no_invariant` annotations frequently — but unlike friends, derived classes are not part of the base class's definition and cannot be enumerated in advance. Potential answers to this question can lead to an explosion in complexity of this feature to opt out.

The primary disadvantage is annotation burden: in codebases with many friend functions or container-heavy algorithms, the number of `no_invariant` annotations might become significant. Whether this tradeoff is acceptable depends on how tight the default bubble can be made without requiring too many opt-outs in common code patterns.

The secondary disadvantage is handling cases where generic code not directly related to the privileged code written by the class owner and their friends is expected to have boundaries where invariants are checked within it. If we make bubbles coarse enough for that to be the case we need a different solution when objects must be passed through such generic code, such as Standard algorithms. We discuss a potential solution to that issue in Section 3.5.14.

3.4.5 Usual Assertions and Invariants

In practice, what programmers most want to express about their types is often broader than a strict invariant. The *usual assertions* of a class describe the properties we expect to hold in normal use — for example, that a `ConstantSumGame` has at least one player, that the scores are within some reasonable range, or that the game name is non-empty. These properties are a superset of the invariants: a moved-from `ConstantSumGame` might have an empty `d_scores` map and a zero `d_totalPoints` (satisfying the invariant) but no players and no name (violating the usual assertions).

This distinction motivates the opt-out pattern. Rather than declaring only the strict invariant — which must hold even for moved-from objects — a programmer could declare the usual, broader set of assertions and then use `no_invariant` on the small number of operations that are only required to

maintain the strict invariant. Move operations, for example, would be annotated with `no_invariant` on the source parameter, indicating that the moved-from object need only satisfy the strict invariant, not the usual assertions.

One might imagine supporting multiple classes of invariant declarations — perhaps `invariant` for the strict invariant and a separate declaration for usual assertions, with different checking rules for each. However, this introduces substantial complexity in specification, implementation, and user mental model with unclear benefit over the combination of a single `invariant` declaration and explicit opt-out annotations. Whether this tradeoff is worthwhile remains an open question.

3.5 When to Assert Invariants

Multiple strategies exist for defining where invariants are checked, varying in how tightly they draw the bubble:

- A *broad* bubble that precisely identifies all the places where checking should be suppressed — internal calls, friend operations, etc. — so that the remaining checking points are exactly right.
- A *tight* bubble that checks aggressively and provides explicit opt-out mechanisms (Section 3.4.4) for cases where the default is too restrictive.
- A *fine-grained* bubble that checks at all function call boundaries regardless of access control (Section 3.5.13).

These approaches are not mutually exclusive: even a broad bubble will likely need some manual annotation for edge cases, and even a tight bubble needs a clear definition of what "tight" means.

In this section we explore what each of these approaches looks like in practice, identifying both the cases they handle naturally and those where they struggle.

As defined in Section 3.1, the bubble is the region of execution within which an object's invariants may temporarily be violated, and its boundaries are the points where invariants are expected to hold.

A key additional observation is that the bubble is inherently stack-related: once an object enters the bubble (e.g., by being passed to a member function), it remains inside the bubble until execution returns back through the boundary where it entered. This means that if an object is already inside the bubble at a higher level on the call stack, a subsequent call into the same object's public interface from within the bubble does not represent a fresh boundary crossing — the object was already in the bubble. The stack-based nature of the bubble is the fundamental insight that resolves the redundant-checking problem for direct internal calls, but it is also what makes the reentrancy case subtle: a callback from external code *does* represent the object leaving and re-entering the bubble, even though the original privileged frame is still on the stack.

There are additional complexities with threads, fibers, and coroutines — where the notion of "the stack" becomes less straightforward — but even the single-threaded case requires careful reasoning about nested bubble entry. Coroutines especially make it clear that any stateful thread-local tracking of being "in" the bubble will eventually fail (in addition to being inordinately expensive), and so any approach needs to find a solution that can be determined statically and not dynamically.

If we can precisely identify these boundary crossings, and we check invariants at every such crossing, we achieve a meaningful guarantee: any code that interacts with the object through its public interface can rely on the invariant being true, and any violation of the invariant will be detected at the point where the object is next observed by code that depends on it.

3.5.1 Identifying Boundary Crossings

The obvious case is calling a public member function on an object from outside the class:

```
void clientCode(ConstantSumGame& g) {
  g.transfer("Alice", "Bob", 10);
    // invariant checked on entry and exit
}
```

But this is far from the only place where an object crosses the encapsulation boundary. Consider all the ways in which a reference to an object can pass from unprivileged code to code that has access to the object's internals:

- **Public member function invocation.** The most obvious case is the invocation of `transfer` on `g`, where `transfer` is public and the caller does not have privileged access to `ConstantSumGame`.
- **Passing an object to a privileged static or friend function.** If `merge` is a static member of `ConstantSumGame`, and it expects two rvalue references to games to be passed to it, then invoking it transitions both of those games from an unprivileged to a privileged context:

  ```
  void caller(ConstantSumGame& a,
              ConstantSumGame& b) {
    auto combined = ConstantSumGame::merge(std::move(a),
                                           std::move(b));
      // both a and b cross into privileged context
  }
  ```

- **Passing an object as a non-this argument to a member function.** If `x = std::move(y)` is called and both `x` and `y` are `ConstantSumGame` objects, then `y` is also crossing into privileged context even though the assignment operator is being called on `x`:

  ```
  void caller(ConstantSumGame& x,
              ConstantSumGame& y) {
    x = std::move(y);  // both x and y enter the bubble
  }
  ```

  This case is particularly important for move operations. As discussed in Section 3.3, a naive move of `ConstantSumGame` moves `d_scores` (emptying the source) but leaves `d_totalPoints` unchanged, breaking the invariant on the source object. Checking invariants on the moved-from object when the function returns ensures that the implementation has properly reestablished those invariants.

- **Returning a reference from a member function.** If a member function returns a reference to a member object of the same type, that object transitions from privileged to unprivileged context.

- **Protected member access from a derived class.** When a derived class calls a protected member of its base, the base object transitions into a context with a different level of access.

Each of these cases represents a point where we might check invariants. The question is whether we can specify the rule precisely enough to be implementable.

3.5.2 Access Control Layers

C++ has three levels of access control: `public`, `protected`, and `private`. Each level defines a different boundary, and it is natural to want invariants at each level. To illustrate, imagine extending `ConstantSumGame` with a protected interface for derived classes (and making some of the public members of `ConstantSumGame` virtual):

```
class ConstantSumGame {
  // ...
protected:
  invariant(d_totalPoints >= 0);
    // derived classes must maintain non-negative total

public:
  invariant(sumOfScores() == d_totalPoints);

  virtual void transfer(std::string_view player1,
                        std::string_view player2,
                        int points);

  // ...
};

class RankedGame : public ConstantSumGame {
  std::vector<std::string> d_ranking;

public:
  invariant(d_ranking.size() == playerCount());
    // ranking must track all players

  void updateRanking();
    // accesses protected members of base

  void transfer(std::string_view player1,
                std::string_view player2,
                int points) override;

  // ...
};
```

Each access level defines a different checking boundary:

- A public invariant should hold whenever the object is observed by code with only public access. Public invariants are the most common case.
- A protected invariant should hold at the boundary between public/protected access and

private access. This allows a base class to specify conditions that derived classes must maintain — in our example, the `d_totalPoints >= 0` constraint that `RankedGame` must preserve when modifying scores through the protected interface.

- A private invariant should hold at all function boundaries, including private member functions. Private invariants provide the strongest guarantee and the most expensive to check.

The access level of an invariant declaration (determined by its position relative to access specifier labels in the class) naturally maps to these different checking boundaries. A public invariant is checked on crossings between code without privileged access and code with any privileged access. A private invariant is checked more broadly.

This layering raises open questions:

- Should a public invariant be checked when crossing from public to protected access (e.g., a derived class calling a protected base member)? Or only at the public/non-public boundary?
- If a function has both public and private invariants, in what order are they checked?
- Can an invariant declared at one access level reference members at a lower or higher access level? A public invariant that calls private member functions creates a circularity risk, but a public invariant restricted to only the public interface cannot express many useful conditions.

3.5.3 Direct Internal Calls

The simplest case within many potential bubbles is to consider whether we treat any public member function that calls another public member function on the same object as a boundary between bubbles.

In `ConstantSumGame`, consider a `resetScores` function that zeroes each player's score by delegating to the public `transfer` member:

```
void ConstantSumGame::resetScores() {
  for (auto& [player, score] : d_scores) {
    if (player != "house") {
      transfer(player, "house", score);
        // calls public member function
    }
  }
}
```

When `resetScores` calls `transfer`, the invariant holds at each intermediate step — this is one of the simple cases. But if invariants are checked on entry to every public member function regardless of the calling context, then `resetScores` pays the cost of checking the invariant on every call to `transfer` — even though `resetScores` is already inside the bubble. Redundant checking is precisely the overhead explosion that Midori encountered.

Because the bubble could be stack-related, one natural answer is that invariant checking should not occur when the call originates from code that is already within the bubble for the same object. The object entered the bubble when the outermost privileged call began, and it will not leave until that call returns. However, this approach — while it eliminates redundant checking — also

suppresses checking at call boundaries that might be useful for detecting bugs within the class's own implementation.

3.5.4 Private Helper Functions

Private member functions are part of the implementation and generally do not need to maintain invariants. For example, an `adjustScore` helper might update a single player's score without touching `d_totalPoints`, relying on the calling function to restore the invariant:

```
void ConstantSumGame::adjustScore(                  // private
    std::string_view player, int delta) {
  d_scores[player] += delta;
  // invariant does NOT hold on return from this function:
  // sumOfScores() != d_totalPoints
}

void ConstantSumGame::addPoints(
    std::string_view player, int points) {
  adjustScore(player, points);
  d_totalPoints += points;
  // invariant restored
}
```

The private-helper pattern is straightforward when the private function is called from a member of the same class. But what if a friend function calls a private function? What if a derived class somehow obtains access? The rules about when private helpers can assume the invariant is broken depend on how we define the bubble.

3.5.5 Callbacks and Reentrancy

The most challenging case is when privileged code invokes external code that calls *back* into the same object. The `zeroSum` operation from our running example (Section 3.3) illustrates this: its lambda captures `this` and is invoked by `std::for_each` while the invariant is broken. A simpler variant makes the issue even more explicit:

```
void ConstantSumGame::forEachPlayer(
    std::function<void(std::string_view,
                       int)> callback) {
  for (auto& [player, score] : d_scores)
    callback(player, score);
}

void ConstantSumGame::taxAll(int taxPerPlayer) {
  int collected = 0;
  forEachPlayer(
    [&](std::string_view player, int score) {
      d_scores[player] -= taxPerPlayer;
      collected += taxPerPlayer;
    });
  // invariant broken during loop:
  // sumOfScores() != d_totalPoints
```

```
    d_totalPoints -= collected;
    // invariant restored
}
```

During the loop, the lambda modifies `d_scores` entries without adjusting `d_totalPoints`, so the invariant does not hold. If `forEachPlayer` checks invariants on its own entry (it is a public member function), or if the callback calls any other member function on the same object — such as `name()` for logging — the broken invariant would be observed. The object has not left the bubble — `taxAll` is still on the call stack — but the callback arrives through `forEachPlayer`, which has no knowledge of the invariant being temporarily violated.

The callback scenario is the reentrancy case, and it represents a genuine category of bugs: code that re-enters an object while it is in an inconsistent state can produce incorrect results or exhibit undefined behavior. Ideally, invariant checking would detect this. However, this is also exactly the kind of situation where brute-force checking is expensive and where false positives can arise from legitimate internal operations.

One might imagine addressing this with a per-object flag that tracks whether an object is already inside a privileged call and suppresses invariant checking for nested entries. As noted in Section 2, D's documentation has at times suggested such a mechanism, but in practice the current D compiler does not implement it — and even if it did, such a flag is not viable for C++: it requires per-object storage overhead that violates the zero-overhead principle underlying [P2900R14], and it would constitute an ABI change for any type that adds an invariant. Any solution we adopt must not impose runtime or storage costs on objects that do not use invariants, nor alter the layout of objects that do.

The opt-out mechanism described in Section 3.4.4 provides one possible approach: if the bubble is defined tightly enough to check on reentrant calls by default, then code that legitimately calls back into an object during an intermediate state can use `no_invariant` to suppress the check at that specific point.

3.5.6 Static Members and Friend Functions Operating on Multiple Objects

A privileged function might operate on multiple objects of the same type, temporarily breaking invariants on all of them. The `merge` function from our running example (Section 3.3) illustrates this: it operates on `lhs`, `rhs`, and a locally-constructed `output`, breaking invariants on each at different points. Even a simpler operation has the same structure:

```
// static member - has private access
void ConstantSumGame::transferBetweenGames(
    ConstantSumGame& from,
    ConstantSumGame& to,
    int points) {
  from.d_totalPoints -= points;
    // from's invariant is now broken
  to.d_totalPoints += points;
    // to's invariant is now broken
  // ... move players/scores to rebalance ...
    // both invariants restored
}
```

At various points during `transferBetweenGames`, one or both objects have broken invariants. When the function returns, all invariants must hold again for every object that was touched.

This example illustrates a fundamental difficulty: the bubble is not just about a single entry and exit point. A function can hold multiple objects in a broken-invariant state simultaneously, and we need a way to express that those objects' invariants need not hold until the function completes.

3.5.7 Concerns Specific to `const` Member Functions

Invariants are almost certainly going to be naturally expressed in terms of the object's accessor functions — calling `const` member functions such as `size()`, `empty()`, `begin()`, `end()`, and so on. This creates several complications:

- If we check invariants on entry to `const` member functions, and the invariant itself calls `const` member functions, we risk infinite recursion.
- `const` member functions should not, in principle, be able to *break* an invariant (since they cannot modify observable state). This argues for not checking invariants on `const` member functions, as they add cost with little benefit.
- However, `mutable` members complicate this picture — a `const` member function that updates a cache stored in a `mutable` member might briefly invalidate an invariant about that cache — and checking such invariants becomes important to find bugs in the internally non-`const` behavior related to `mutable` members.
- More subtly, if we do not check invariants on `const` member functions, then a member function that temporarily breaks the invariant can safely call `const` members for its own internal logic. In our running example, `transfer` calls `name()` for logging while the invariant is broken (Section 3.3). If `const` member functions trigger invariant checks, this internal logging would fire a false positive. The false-positive risk is a significant practical concern, as many implementations naturally call their own accessors.

Whether `const` member functions should be subject to invariant checking is an open question, and the right answer might depend on the access level and cost of the invariant. Two distinct aspects of this question must not be conflated:

- Whether we *expect* an invariant to be true on entry to and exit from `const` member functions — this is a semantic question about what the invariant means, and would be controlled by a specifier on the invariant declaration itself.
- Whether we choose to *evaluate* an invariant on `const` member function boundaries in a given build — this is a cost question, and is the domain of the compiler's build configuration and labels (Section 3.6.2).

The first determines correctness; the second determines overhead. A `const`-applicable invariant that is too expensive to check on every `const` call can still be declared as one that *should* hold there, while using a label to control how often it is actually evaluated.

3.5.8 Constructors and Destructors

The common understanding of a class invariant is that it should be *established* by all constructors and should remain true for the lifetime of the object until it is destroyed. This maps naturally to:

- An invariant is a postcondition of every constructor. The constructor's job is to bring the object from raw memory into a state that satisfies the invariant.
- An invariant is a precondition of the destructor. The destructor can assume the invariant holds when it begins execution. (It need not *maintain* the invariant — after the destructor completes, the object no longer exists.)

This basic model is straightforward but raises several detailed questions:

- **Delegating constructors.** When one constructor delegates to another, the invariant should hold after the *outermost* constructor completes, but not necessarily after each intermediate delegation. The delegated-to constructor may leave the object in a state that the delegating constructor will further modify. For example, `ConstantSumGame` might have a constructor that creates a game with initial players:

  ```
  ConstantSumGame::ConstantSumGame(
      std::string name,
      std::vector<std::string> players,
      int startingPoints)
  : ConstantSumGame(std::move(name))
      // delegates: invariant holds (0 == 0)
  {
    d_totalPoints = startingPoints * players.size();
      // invariant now broken
    for (auto& p : players)
      d_scores[p] = startingPoints;
      // invariant restored
  }
  ```

  Between the delegating constructor's return and the completion of the outer constructor, the invariant need not hold.

- **Partially-constructed objects.** If a constructor throws an exception, the object is never fully constructed and the invariant need not have been established. Member subobjects that were constructed will be destroyed, but the enclosing object's invariant was never promised to hold.
- **Destruction during stack unwinding.** If a destructor is invoked during stack unwinding due to an exception, the invariant should still hold as a precondition — the exception that caused unwinding is not an excuse for leaving objects in an invalid state. This is consistent with the C++ guarantee that destructors of fully-constructed objects are always called.

3.5.9 Trivial Special Member Functions

In C++, trivial special member functions (SMFs) — trivial default constructors, copy/move constructors, copy/move assignment operators, and trivial destructors — currently cannot have function

contract assertions. This creates a tension with class invariants: if an invariant is a postcondition of every constructor and a precondition of the destructor, then any type with a trivial destructor or trivial copy constructor would seem to be unable to declare invariants at all.

The general interaction between contract assertions and trivial SMFs was explored in [P2932R3], and the result of that discussion was that [P2900R14] proposed that contract assertions cannot be placed on trivial SMFs. Invariants, however, depending on their design, might implicitly put assertions on a trivial SMF and we must therefore decide what the result of that would be.

There are two primary approaches we could take:

- **Prohibit invariants on types with trivial SMFs.** This is the simplest rule but severely limits the applicability of invariants. Many lightweight value types have trivial operations but would benefit from invariant checking. In particular, trivial relocation (as proposed by [P2786R13]) is effectively an SMF that is trivial for a great number of types that also have invariants. In general, if we ever expect to add new SMFs to the language this approach would be infeasible, as large numbers of types in the wild would already exist that have invariants and would also have a trivial instance of the new SMF.
- **Allow contract assertions on trivial SMFs with the constraint that they are always ignorable.** The SMF remains trivial, but we allow the invariant to be skipped (preventing, for example, the use of labels that don't allow for the *ignore* semantic). This allowance is needed in order to handle the cases where, due to type erasure or the nondeterministic nature of when some trivial SMFs are considered to be evaluated there is no way for the compiler to identify that the trivial SMF has executed at all.

One could also consider invariants forcing all SMFs to be non-trivial by merely being present in the class, but that would violate the prime directive of [P2900R14], and does not warrant any further discussion.

The interaction between invariants and triviality is an important practical concern that must be resolved before a concrete proposal can be made.

3.5.10 Virtual Functions and Inheritance

When a class hierarchy uses invariants, the relationship between base and derived class invariants must be defined. Extending our running example, consider the `RankedGame` we discussed in Section 3.5.2.

In C++, a derived class's base-class subobject is always an instance of the base type — it can be bound to a reference, passed to functions expecting the base type, and destroyed as a base object. Because the base subobject remains a valid object of its base type throughout its lifetime, the base class's invariants must apply to it unconditionally. `RankedGame` must maintain `sumOfScores() == d_totalPoints` in addition to its own ranking invariant. A derived class can *add* invariants of its own, but it cannot weaken or remove invariants established by its base.

For virtual functions called through a base-class pointer or reference, the question is which invariants are checked. If we call `transfer` on a `ConstantSumGame&` that actually refers to a `RankedGame`, should only the base invariant be checked, or also the ranking invariant? The virtual-function design (approved for C++29) in [P3097R3] provides part of a natural model for this — by adding implicit preconditions and postconditions to the base class's virtual functions a base class's invariants

implicitly become part of that caller-facing interface. On the other hand, a derived class's final overrider has protected access to the base class data members, and thus might have callee-facing checks of some invariants from the base class due to entering a privileged bubble with access to the base class's internals.

In all cases the checking of invariants across a class hierarchy follows from which functions are considered within or outside of the relevant bubble, and which are considered to be boundary crossings.

3.5.11 Containers, Arrays, and Indirect Access

The feature's design must also address whether invariants should be transitively applied to contained members. When a container's invariants are checked, should that check the invariants of all objects in the container? When checking the invariants of a pointer (`T*`) or smart pointer (`std::shared_ptr<T>`) should that include validating the invariants of the referenced object?

On the one hand, transitive checking could be considered unnecessary because the contained (or referenced) object will have to cross a boundary when used, and can have its invariants checked then. On the other hand, if a function is going to process a list of objects and expect them to all be valid, it should be self-evident on that function's interface (which will take the list or a reference to a list as a parameter) that there is an expectation at the time of passing that all objects are valid.

These questions apply equally (in relation to implicit assertions) to builtin types and user-defined types. Whether we expect the invariants to hold whenever passing around objects with such indirect access to other objects depends on whether we aim our design to tackle fine-grain and ubiquitous bubbles or to only define our boundary crossings when an object of a given type explicitly crosses a boundary on its own.

3.5.12 Exceptions

So far, we have focused on boundary crossings related to normal function entry and exit, as well as the normal use of builtin operators to access arrays or dereference pointers. All of these are clear ways in which an object enters or leaves a bubble. Another important factor to consider is when code execution leaves a bubble through stack unwinding by throwing an exception — each object that had entered the bubble and possibly had its invariant broken could be left in a state where it is accessible from other code in that same broken state.

In order to test this scenario, when the stack is unwound we could take the approach that all objects accessible in a function are crossing a bubble boundary and should have their invariants checked whenever unwinding a stack frame or completing a catch clause. Protecting this path of execution from leaking invalid objects to the rest of a program would be a key part of validating that a program is exception-safe.

3.5.13 Fine-Grained Bubbles

An important point should be made about potential designs where we keep bubbles very fine-grained — where any function call or other operation that passes in or returns objects of a type should be considered a boundary crossing unless there is an explicit opt-out.

Such all-boundaries checking produces the tightest possible bubble. Every function call that receives an object would check its invariants on entry and exit, whether the function has privileged access or not. The advantage is that it communicates the invariant across functions that should not themselves be able to break it — a free function that takes a `ConstantSumGame` by reference would verify the invariant on entry, confirming that the caller has not passed a broken object, even though the function has no special access.

Even invariants that are not fully encapsulated — such as those involving public data members — will be caught as bugs by this approach in very close proximity to where the invariant is broken.

The primary disadvantage is that it becomes impossible to pass an object with a broken invariant through a chain of functions that are not aware they need to support a broken-invariant object of that type. Standard algorithms are the canonical example: a `std::sort` on a range of `ConstantSumGame` objects would check invariants on every swap, comparison, and move — even though the algorithm is operating at a level of abstraction that has no knowledge of `ConstantSumGame`'s invariants and may be rearranging objects in intermediate states where invariants do not hold.

This tension between the tightest possible checking and the practical need to pass objects through generic code is a fundamental constraint on how tight the bubble can be.

3.5.14 Invariant-Free Projections

One approach to the generic-code problem raised by fine-grained bubbles is to define an *invariant-free projection* of a class — a view of the object that explicitly does not carry invariant obligations. Code that needs to operate on objects in potentially broken-invariant states would work with this projection instead of the original type.

Such a projection could be a reference-like type that provides access to the same underlying storage but suppresses invariant checking at its boundaries. With C++'s reflection facilities (as they develop), it may be possible to generate such projections mechanically.

However, this approach introduces deep structural complexity. While a container of elements like `std::vector<ConstantSumGame>` might be available, to make use of it when its contents might be in invalid states we would want to pass around a `std::vector<UnderlyingConstantSumGame>` or something similar. This kind of type relationship is hard to make naturally arise in C++, and would require a broad and invasive change to the type system to achieve.

3.6 Controlling Evaluation Semantics

Even if we perfectly specify *when* invariants should hold, the question of when to actually *check* them at runtime remains. The runtime-cost question is partly addressed by the existing Contracts mechanism — evaluation semantics already allow any contract assertion to be *ignored*, *observed*, *enforced*, or *quick-enforced* — but class invariants introduce additional dimensions of control.

3.6.1 Invariants with Varying Complexity

Many useful invariants have non-constant complexity:

```
class SortedVector {
  std::vector<int> data_;
```

```
public:
  invariant(data_.size() <= data_.capacity());          // O(1)
  invariant<audit>(
    std::is_sorted(data_.begin(), data_.end()));        // O(n)
};
```

The `audit` label from [P3400R4] provides an existing mechanism for distinguishing cheap invariants that should be checked broadly from expensive ones that should be checked only in special builds.

3.6.2 Label Extensions for Invariant Control

Beyond `audit`, class invariants might benefit from label facets that do not exist for other contract assertions.

It is important to distinguish between two kinds of control that might superficially seem similar:

- Whether an invariant is *expected to be true* at a particular kind of function boundary — this is a semantic property of the invariant declaration itself, and should be expressed through a specifier on the declaration (e.g., whether the invariant applies on `const` member function boundaries at all).
- Whether an invariant is *evaluated* at a particular boundary in a given build — this is a cost-management question, and is the domain of labels.

Labels lend themselves to the second case, but not the first. It is important that labels (as is made clear in [P3400R4]) do not change the meaning of any contract-related construct, and that things where we change the meaning be done with specifiers that are part of the syntax and outside of the assertion-control object.

What matters for invariants, however, is that we will be applying an invariant with one label in many different contexts. A label, therefore, might want to vary the impact it has on decisions based on each of those individual contexts — such as the type of function or boundary where the invariant is being checked. All of that information would need to be passed to a new form of the computed semantic facet in order to leverage the extra information to fine-tune checking.

4 Open Questions

The following questions must be answered before a concrete proposal for class invariants in C++ can be made:

1. **Can the encapsulation boundary be precisely defined?** Can we specify exactly which function calls constitute boundary crossings in terms of C++'s access control model, including friends, derived classes, and implicit conversions?
2. **How do we handle reentrancy?** When privileged code calls external code that calls back into the same object, should the invariant be checked? If yes, how do we avoid the false positives that arise from legitimate internal operations that temporarily break invariants?
3. **What is the model for friends operating on multiple objects?** How does a friend function express that it has taken multiple objects into its bubble and will restore their invariants

before returning? The `no_invariant` qualifier (Section 3.4.4) is one possible mechanism, but is it sufficient for all cases?

4. **Should `const` member functions trigger invariant checks?** If not by default, what mechanism opts them in? How do we handle `mutable` members?

5. **How do invariants interact with constructors and destructors?** An invariant clearly must hold after a constructor completes and before a destructor begins, but what about delegating constructors, partially-constructed objects, and objects being destroyed during stack unwinding?

6. **How do invariants interact with inheritance?** If a base class declares an invariant, is it automatically checked for derived class member functions? Can a derived class *strengthen* but not weaken the base invariant (as in Eiffel)? What about virtual functions?

7. **What label facets are needed?** Beyond the basic evaluation semantic, what invariant-specific control do we need? How does this interact with the assertion-control framework of [P3400R4]?

8. **What form should manual invariant control take?** Section 3.4.3 proposes a new operation, `assert_invariants`, for explicit checking and Section 3.4.4 proposes a new qualifier `no_invariant` for explicit suppression. Are both needed? Should they be keywords, library facilities, or attributes? How do they interact with each other and with the automatic checking points?

9. **What about trivial special member functions?** Can a type with invariants have a trivial destructor? A trivial copy constructor? If invariants must be checked on destruction entry, does that inherently prevent triviality?

10. **How do invariants interact with concurrency?** If an invariant reads member variables that are modified under a mutex lock, must the invariant check also acquire the lock? If so, methods that check invariants on entry and exit would acquire the mutex twice (or require a recursive mutex). If not, the invariant check races with concurrent modifications.

11. **What guarantee are we actually providing?** If all invariants are checked at all boundary crossings with a checked semantic, what can a user rely on? Is the goal full invariance (modulo undefined behavior elsewhere in the program), or something weaker?

12. **How fine-grained should the bubble be?** Should the bubble be tied to access control (checking only at boundaries between privileged and unprivileged code), or should it extend to all function call boundaries regardless of access (Section 3.5.13)? If the latter, how do we handle objects that must pass through generic code (standard algorithms, containers) in a broken-invariant state?

13. **Should there be a distinction between invariants and usual assertions?** Is it sufficient to have a single `invariant` declaration combined with `no_invariant` opt-outs, or do we need a way to express the broader "usual assertions" of a type separately (Section 3.4.5)?

5 Conclusion

Class invariants are an often-requested extension to C++ Contracts that would greatly aid in leveraging existing software engineering practices to validate correctness in C++ software. The ability to declare, once, a condition that a class maintains throughout its lifetime and have that condition checked at the appropriate boundaries would bring significant value to both new and existing codebases.

Any approach we take must begin by deciding the scope and nature that we are targeting, and then determine how that decision leads to a reasonable set of answers to all of the above questions. Importantly, any such proposal must identify how common usage patterns will interact with attempts to apply invariants, and both what the syntactic burden on working with the design will be and what benefits will come from the checking that is enabled.

The first, highest-level question to answer before proceeding is to select the targeted bubble granularity and porousness, such as:

- Check invariants on all boundaries.
- Check invariants on changes in access level (with or without some form of scoped awareness).
- Check invariants on only member functions of an object.

After that decision we need to identify which tools are needed to work around the limitations of that decision:

- A syntax to manually assert invariants.
- A syntax to opt out of the expectation that invariants hold somewhere they would be implicitly added.
- A tool to avoid checking invariants in generic code where checks for them might be implicitly added.

An invariant design that builds on the Contracts feature in C++26 is certainly possible, but to achieve it we need to come to a wider understanding of the questions that must be answered and the implications of those potential answers.

Acknowledgments

Thanks to Timur Doumler, Herb Sutter, Lucian Radu Teodorescu, Peter Bindels, Thomas Mejstrik, Corentin Jabot, Tony van Eerd, Lisa Lippincott, and Ivan Lazaric for discussions that motivated this paper, as well as (in some cases) reviews of this paper.

Claude (Anthropic) was used for editorial assistance and prior art research during the preparation of this paper.

Bibliography

[Ada2012RM] "Ada Reference Manual, Section 7.3.2: Type Invariants". http://www.ada-auth.org/standards/rm12_w_tc1/html/RM-7-3-2.html, 2012

[DlangInvariantIssue] "Class/struct invariants are plain useless". https://github.com/dlang/dmd/issues/18659, 2024 D Language issue tracker

[Dlang] "D Language Specification: Class Invariants". https://dlang.org/spec/class.html#invariants, 2024

[Duffy2016] Joe Duffy, "The Error Model". http://joeduffyblog.com/2016/02/07/the-error-model/, 2016

[Leino2004] K. Rustan M. Leino and Peter Müller, "Object Invariants in Dynamic Contexts". *ECOOP 2004*, *LNCS*, volume 3086 (Springer, 2004), pp. 491–515

[Leino2010] K. Rustan M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness". *LPAR-16*, *LNCS*, volume 6355 (Springer, 2010), pp. 348–370

[Meyer1997] Bertrand Meyer, *Object-Oriented Software Construction*. 2nd edition (Prentice Hall, 1997) Chapters 11–12

[P1995R1] Joshua Berne, Andrzej Krzemieński, Ryan McDougall, Timur Doumler, and Herb Sutter, "Contracts — Use Cases", 2020
http://wg21.link/P1995R1

[P2755R1] Joshua Berne, Jake Fevold, and John Lakos, "A Bold Plan for a Complete Contracts Facility", 2024
http://wg21.link/P2755R1

[P2786R13] Pablo Halpern, Joshua Berne, Corentin Jabot, Pablo Halpern, and Lori Hughes, "Trivial Relocatability For C++26", 2025
http://wg21.link/P2786R13

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, "Contracts for C++", 2025
http://wg21.link/P2900R14

[P2932R3] Joshua Berne, "A Principled Approach to Open Design Questions for Contracts", 2024
http://wg21.link/P2932R3

[P2961R2] Timur Doumler and Jens Maurer, "A natural syntax for Contracts", 2023
http://wg21.link/P2961R2

[P3097R3] Timur Doumler, Joshua Berne, and Gašper Ažman, "Contracts for C++: Virtual functions", 2026
http://wg21.link/P3097R3

[P3361R1] Esa Pulkkinen, "Class invariants and contract checking philosophy", 2024
http://wg21.link/P3361R1

[P3400R4] Joshua Berne, “Controlling Contract-Assertion Properties”, 2026
http://wg21.link/P3400R4