Contracts for C++: Virtual functions

Timur Doumler (papers@timur.audio)
Joshua Berne (jberne4@bloomberg.com)

Document #: P3097R1
Date: 2025-12-13
Project: Programming Language C++
Audience: EWG

Abstract

We propose an extension to C++26 Contracts that enables specifying precondition and postcondition assertions on virtual functions. In our proposed model, the assertions of an overriding function are independent of those in the overridden function. In a virtual function call, the precondition and postcondition assertions of both the statically chosen function and the final overrider are evaluated. This approach evolves earlier proposals by supporting a broader range of use cases found in existing code, integrating more naturally with C++26 contract-evaluation semantics and contract-violation handling, and providing a better fit for the C++ language than the more restrictive models in languages like Eiffel and D.

Contents

Revision history

R1 (December 2025 mailing):

- Reorganised the paper
- Factored out design space analysis into [P3600R0]
- Rebased proposed wording onto [N5014]

R0 (April 2024 mailing):

- Original version presented to SG21 and EWG

1 Motivation

Runtime polymorphism is a fundamental part of C++. Many libraries and APIs are designed around virtual functions, making it one of the core idioms of the language. Other programming languages with built-in contract assertions, such as Eiffel, D, and Ada, fully integrate these assertions with virtual functions and runtime polymorphism.

The importance of offering similar integration in C++ has been recognised repeatedly throughout the history of C++ Contracts standardisation (e.g., [N4110], [P0147R0], [P0247R0], [P3173R0]). Designing such integration, however, has proven difficult ([P2899R1] Section 3.3.2). Multiple approaches to precondition and postcondition assertions on virtual functions have been proposed, gained consensus, and lost it again when fundamental issues were found (e.g., [P0542R5], [P2521R5], [P2954R0], [P2932R3]).

The design presented in this paper addresses the shortcomings of earlier proposals, supports a broader range of use cases found in real-world code, and aligns naturally with C++26 contract-evaluation semantics and contract-violation handling. A previous revision of this paper — [P3097R0] — gained strong consensus in SG21 and EWG,[1] was merged into the main C++ Contracts proposal [P2900R13], and underwent thorough CWG wording review.

However, in [P3506R0] and [P3573R0] concerns were raised that our approach diverged from the established model in Eiffel and D and lacked sufficient deployment and usage experience. These concerns convinced EWG to remove the feature from the C++ Contracts paper. EWG was unable to resolve this disagreement in time for C++26, and instead decided to ship contract assertions in C++26 without support for virtual functions ([P2900R14]).

To help re-establish consensus and advance this important feature, in [P3600R0] we have conducted the most comprehensive analysis to date of the use cases, available design space, and tradeoffs among the many possible designs for enabling precondition and postcondition assertions on virtual functions.

Based on this extensive work, we are more confident than ever that the design presented in [P3097R0] and merged into [P2900R13] is the right solution for C++. In this revision, we therefore re-propose that design, with wording rebased on the current C++26 working draft [N5014] and a complete implementation of that wording available in GCC.

2 Overview

We propose to allow specifying precondition and postcondition assertions on virtual functions, in the same way they can already be specified on non-virtual functions:

[1] EWG poll from St. Louis (June 2024): P3097R0 — Contracts for C++: Support for Virtual Functions, we would like to see this paper merged into P2900 and progress contracts with virtual function support. SF/F/N/A/SA 18/15/5/1/2

```
struct Car {
  virtual void drive(float speed)
    pre (speed <= 100); // don't go too fast!
};

void testCar {
  Car car;
  car.drive(90);    // OK
  car.drive(120);   // contract violation - going too fast
}
```

Following [P2900R14] Design Principle 10, precondition and postcondition assertions are *not inherited* by overriding functions. If an overriding function specifies no precondition or postcondition assertions, then no such assertions apply to *that* function, regardless of whether the overridden function specifies any. An overriding function may also specify wider or narrower precondition and postcondition assertions than the function it overrides:

```
struct HyperCar : Car {
  void drive(float speed) override
    pre (isCharged());  // needs to be charged before driving

  void charge();
  bool isCharged() const;
};

void testHyperCar() {
  HyperCar hyperCar;
  hyperCar.drive(90);   // contract violation - forgot to charge

  hyperCar.charge();
  hyperCar.drive(120);  // OK - can go arbitrarily fast when charged!
}
```

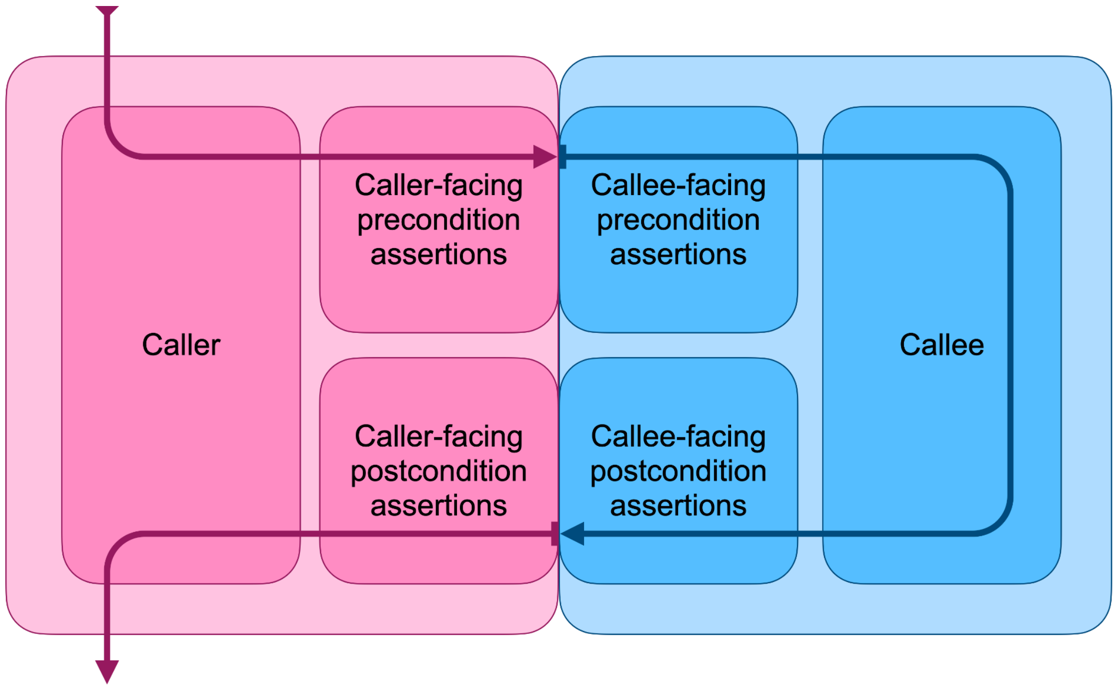
However, as part of any function call, *two sets* of assertions are always evaluated:

- The *caller-facing assertions* — those of the function named by the call expression,
- The *callee-facing assertions* — those of the function actually invoked.

These are evaluated in the following sequence:

- Evaluate the caller-facing precondition assertions,
- Evaluate the callee-facing precondition assertions,
- Execute the body of the function,
- Evaluate the callee-facing postcondition assertions,
- Evaluate the caller-facing postcondition assertions.

This sequence is illustrated in the following diagram:


Caller
Caller-facing precondition assertions
Callee-facing precondition assertions
Callee
Caller-facing postcondition assertions
Callee-facing postcondition assertions

Depending on the kind of function call, the caller-facing and callee-facing assertions may be identical (and can be deduplicated, see [P2900R14] Section 3.5.7), or they may differ.

In a *virtual function call*, they typically differ: the function named by the caller (the *statically chosen function*) is in general not the function actually invoked by dynamic dispatch (the *final overrider*). According to the sequence above, the precondition and postcondition assertions of *both functions* are evaluated:

```
void testPolymorphicCar(Car& car) {
  car.drive(90);    // test 1
  car.drive(120);   // test 2
}

int main() {
  Car car;
  testPolymorphicCar(car);      // test 1 succeeds, test 2 fails

  HyperCar hyperCar;
  testPolymorphicCar(hyperCar); // both tests fail
  hyperCar.charge();
  testPolymorphicCar(hyperCar); // both tests succeed
}
```

In a call through a *function pointer* — including a pointer-to-member-function that invokes a virtual function — the caller-facing assertions are an empty set, since pointers cannot have assertions associated with them ([P3327R0]). Nevertheless, the sequence above ensures that the precondition and postcondition assertions of the function actually invoked are still evaluated.

In all other cases — including a fully qualified, non-virtual call to a virtual function — the caller-facing and callee-facing assertions are identical.

This design deliberately differs from earlier C++ proposals where contract assertions were always inherited by overriding functions. It also deliberately diverges from Eiffel and D, where satisfying the preconditions of any overridden function is sufficient to call a virtual function (i.e., preconditions can only be widened, but not narrowed) and the postconditions of *all* overridden functions are always applied (i.e., postconditions can only be narrowed, but not widened). In the remainder of this paper, we describe and motivate these design choices and mention the alternatives considered.

A more detailed, systematic analysis of the use cases for precondition and postcondition assertions on virtual functions, the available design space for this feature, and the tradeoffs among the many possible designs within that space — including all designs known from other programming languages and earlier C++ proposals — is available in [P3600R0].

3 Design goals and principles

3.1 Substitution principle and program correctness

Contract assertions allow the user to express and optionally verify expectations on a program's correctness, helping identify violations of those expectations as bugs. Extending the C++ facility to virtual functions requires understanding the kinds of correctness expectations that commonly apply to such functions, and ensuring that the proposed extension can express those expectations effectively.

Some expectations are specific to an individual function or the particular dynamic type that it is a member of. Others arise from that type's participation in an inheritance hierarchy. Such hierarchies are commonly expected to follow the *substitution principle*: any correct use of an object of polymorphic type should remain correct if the dynamic type of that object is a derived class.

Consider a virtual function call: a caller invokes a *statically chosen function*, and dynamic dispatch selects a *final overrider*. The statically chosen function has a (plain-language) contract consisting of two parts:

- *preconditions* that the caller is expected to satisfy when calling the function,
- *postconditions* that the function is expected to establish when returning to the caller.

The caller is correct if it satisfies the preconditions of the statically chosen function. The final overrider is correct if — assuming the caller has met the preconditions of its statically chosen function — it establishes that statically chosen function's postconditions, or, if unable to do so, triggers an appropriate error-handling mechanism (e.g., throws an exception that the caller can handle). Any other outcome constitutes a bug.

The (plain-language) contract of a function may include both explicit and implicit expectations, and some may not be verifiable within the C++ abstract machine. Consequently, only a *subset* of those expectations can be expressed using precondition and postcondition assertions. While this subset will remain a proper subset in most cases, expanding the expressible subset is the fundamental goal of any extension to C++26 contract assertions.

For virtual functions, the expressible subset is currently very limited: `pre` and `post` cannot be used at all, leaving only `contract_assert`. The latter is confined to the function body and thus cannot capture conditions that relate to the caller-callee boundary, which is particularly important for virtual function calls. This proposal removes this limitation.

3.2 Adoptability in legacy code

C++ is a mature and widely used language with billions of lines of existing code deployed across many diverse domains, including those important for the functioning of human society. As a multi-paradigm language, it also encompasses a wide variety of programming styles and coding guidelines. C++26 contract assertions were explicitly designed for broad adoptability across these code bases. For this goal to succeed, the feature must meet several key requirements.

Consider a function in a pre-C++26 codebase. To incrementally improve the correctness of this codebase, we may wish to add a contract assertion that expresses some part of that function's existing plain-language contract. When doing so:

- Any existing, *correct* use of the function must remain well-formed and correct,
- Any contract violation reported must correspond to a genuine violation of the function's (plain-language) contract.

In other words, contract assertions must not break an existing program. While false negatives are inevitable, *false positives must never be forced on users by the language design.*

These requirements are not unique to contract checking; they apply equally to any tool intended to detect bugs in existing code, such as compiler warnings or static analysis tools.

The same principles govern the semantics of precondition and postcondition assertions on virtual functions. As discussed in the previous section, correctness in the context of inheritance and runtime polymorphism introduces additional considerations. The proposed extension must therefore both:

- Maximise the user's ability to diagnose *incorrect* uses and implementations of virtual functions, and
- Guarantee that *correct* uses and implementations of virtual functions never produce spurious contract violations, compilation errors, or other breakage when assertions are added — except in cases where the assertion itself is defective, in which case the issue should be easy for the user to identify.

3.3 Deployability in modular codebases

Very few C++ programs can be completely compiled from source. Dependencies are often provided as headers and precompiled binaries. While ABI compatibility generally ensures that linking against such binaries works correctly, it is inevitable that some binaries will be built with a pre-C++26 compiler, others with a compiler supporting C++26 contract assertions, and still others with newer compilers implementing extensions such as the one proposed here.

Further, even when large parts of a program can be rebuilt from source, that source code is frequently owned by different entities. Modifying dependencies — even when their source is available — may require maintaining a separate fork, which is often impractical. Importantly, inheritance hierarchies can span across such component boundaries. A base class may be defined in a library header and derived from another component, or independently in multiple components.

These realities impose design constraints on how precondition and postcondition assertions on virtual functions must behave. For the feature to be deployable at scale, different components should be able to introduce contract assertions independently, without requiring coordinated changes across the entire inheritance hierarchy or the codebases that participate in it.

3.4 Lessons from other programming languages

Any new C++ feature should be informed by prior art. We are aware of three languages that provide built-in precondition and postcondition assertions fully integrated with runtime polymorphism: Eiffel, D, and Ada.

For a detailed description of their respective designs, see [P3600R0]. Each offers valuable lessons, but none provides a model suitable for direct adoption in C++, owing to fundamental differences in language design and philosophy.

Eiffel, designed by Bertrand Meyer in the 1980s, is not widely used today but remains historically significant for introducing contracts-based programming. It emphasises reliability, is highly constrained, garbage-collected, and memory-safe. By contrast, C++ is a general-purpose language with many low-level facilities and unbounded undefined behaviour. Avoiding undefined behaviour during contract checking is therefore an important design consideration for C++, but not for Eiffel, which precludes undefined behaviour by construction.

Further, Eiffel adheres to a formally defined, pure object-oriented paradigm. By contrast, C++ is a multi-paradigm language. Unlike Eiffel, C++ must accommodate adding assertions to existing code organised across independently developed components, and cannot reject *correct* code simply because it does not fit a particular programming paradigm.

Even within the object-oriented paradigm, C++'s object model differs markedly from Eiffel's. Meyer himself often criticised these differences,[2] noting where C++ diverged — sometimes sharply — from the object-oriented principles underpinning Eiffel.

The **D** programming language, released in 2001, also included contract assertions from the outset. Aside from syntax and other superficial aspects, D's approach — especially its integration of precondition and postcondition assertions with virtual functions — closely resembles Eiffel's. This is noteworthy because D is otherwise much closer to C++ in spirit and design. While D's contract assertions are well-documented and actively supported in the compiler, there is little evidence that they are widely used or measurably improve outcomes in D codebases. D's overall adoption remains limited.

Ada, by contrast, enjoys significant use in safety- and mission-critical domains such as aerospace, defence, transportation, and medical systems. While Ada's first implementation dates back to 1983, contract assertions were introduced in Ada 2012 and are now an established and widely used part of the language.

Ada's integration of precondition and postcondition assertions with runtime polymorphism is notably complex. It distinguishes *specific* and *class-wide* precondition and postcondition assertions — spelled differently in the language — with non-trivial rules governing which are evaluated and when. Although this design provides flexibility, the resulting ergonomics and learning curve do not seem appropriate for C++.

In addition, Ada's model of runtime polymorphism differs substantially from C++. The language lacks virtual member functions and other familiar C++ constructs altogether; instead, the feature is built on Ada-specific concepts such as tagged types, packages, and attributes, which have no direct C++ analogues.

Attempting to translate these mechanisms into C++ seems unlikely to produce a feature that integrates seamlessly into the language. Instead, we should design the feature natively within C++ from the ground up. As shown in Section 5, such a design can offer the full range of functionality available in Ada.

Other programming languages either lack precondition and postcondition assertions as an established language feature, do not integrate them meaningfully with runtime polymorphism, or focus exclusively on static correctness proofs and annotation-based formal specifications rather than runtime checking and tooling support as C++26 does.

[2] For example, in [OOSC2], Meyer described C++'s decision to allow both virtual and non-virtual member functions, with non-virtual as the default, as "regrettable" and "damaging to software development". He characterised C++ as a transition technology and predicted it would not "remain a major tool for the software engineering community well into the twenty-first century, as it would then be overstaying its welcome" — an outcome history has disproven.

4 Discussion

4.1 Syntax

In a virtual function declaration, the sequence of precondition and postcondition assertions appears after *virt-specifier*s (`override`, `final`) but before a *pure-specifier* (`= 0`).

The contract-assertion syntax has been designed from the outset with virtual functions in mind ([P2885R3], [P2961R2]). Consequently, the grammar adopted for C++26 already accommodates this placement and does not require any changes.

4.2 Widening preconditions, narrowing postconditions

Most earlier C++ proposals required precondition and postcondition assertions on virtual functions to be identical across the entire inheritance hierarchy; they differed only in how this constraint was enforced.[3] However, such a design is too restrictive to be useful in practice. Consider this example from [P0247R0]:

```
class Display {
public:
  virtual void post_message(std::string_view s)
    pre (is_ascii(s)) = 0;
};

class XDisplay : public Display {
public:
  void post_message(std::string_view s) override
    pre (is_utf8(s));
};
```

The abstract base class `Display` requires ASCII strings. The derived class `XDisplay`, however, has a wider precondition: it can handle UTF-8. A program that uses `XDisplay` directly — without relying on the `Display` interface — should be able to display both ASCII and UTF-8 messages without causing a contract violation.

Now, consider an abstract class for generating sequences of integers by repeatedly calling a virtual function:

```
struct Generator {
  virtual int next() = 0;
};
```

[3] [P2954R0] proposed that overriding functions shall specify no precondition and postcondition assertions, and instead implicitly inherit them from the overriding function. [N4415], [P0287R0], [P0380R0], [P2521R5] instead required that overriding functions shall always repeat the sequence from the overridden function. [P0380R0], [P0542R5], [P2388R4] made this repetition optional.

Derived classes can implement different kinds of generators. Each should be able to verify its own implementation through postcondition assertions specific to that class. For example, a generator that always returns the same user-specified constant value might look like this:

```
 struct ConstantGenerator : Generator {
   void setValue(int value);
   int getValue();
   int next() override
     post (r: r == getValue());
};
```

`ConstantGenerator::next` has a narrower postcondition than its overridden function — it provides a stronger guarantee. A postcondition assertion that specifies this property should compile, and be evaluated whenever `ConstantGenerator::next` is called.

Such *widening of preconditions* and *narrowing of postconditions* is a basic application of contracts-based programming, fully compatible with the substitution principle, and supported by Eiffel, D, and Ada. Our proposal likewise enables both use cases.

4.3 Caller-facing and callee-facing assertions

The proposed sequence for evaluating caller-facing and callee-facing assertions is designed to be both intuitive and comprehensive. It can be easily taught as follows: "the caller calls one function, and dynamic dispatch selects another — we check the precondition and postcondition assertions of *both*".

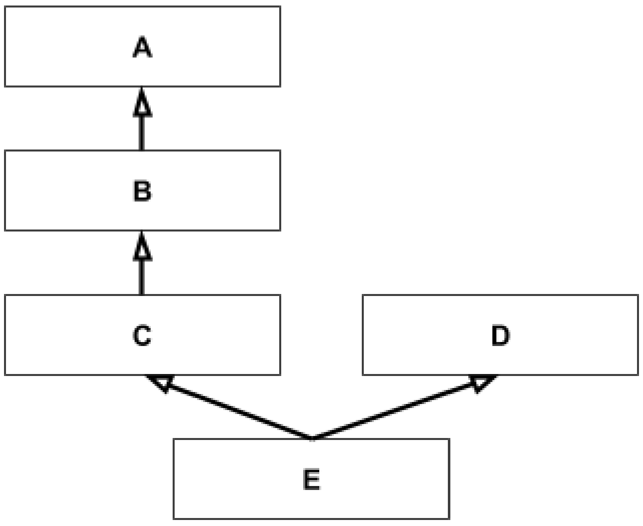
This model naturally captures all four classes of bugs that can manifest at the caller/callee boundary of a virtual function call:

- **Caller-side bugs:**
    - A **caller-facing precondition** violation indicates incorrect use of the polymorphic interface.
- **Callee-side bugs:**
    - A **callee-facing postcondition** violation indicates a bug internal to the implementation of the derived class.
- **Substitutability bugs:**
    - A **callee-facing precondition** violation indicates that the derived class failed to serve a correct call of the polymorphic interface it implements;
    - A **caller-facing postcondition** violation indicates that the derived class failed to provide the guarantees that the caller expects of the polymorphic interface.

No other known design — whether proposed for C++ or implemented in other programming languages — covers all four of these "quadrants".

Eiffel and D implement widening of preconditions by OR-ing the precondition assertions of the final overrider with those of *all* overridden functions, which removes the caller-facing precondition check. In Ada, *specific* precondition and postcondition assertions are evaluated only for the final overrider, omitting caller-facing checks entirely. Conversely, its *class-wide* precondition assertions omit the callee-facing precondition check; there is no direct way to express a precondition assertion that should be evaluated both for a statically chosen function and for a final overrider selected by dynamic dispatch.

At the same time, our proposal evaluates *only* those assertions that are directly relevant to a given virtual function call: those of the statically chosen function and the final overrider. Consider the following class hierarchy:



If each class in this hierarchy defines a virtual function `void f()`, and a caller invokes this function via a virtual call through a pointer or reference to class `B`, while the dynamic type of the object happens to be `E`, then the precondition and postcondition assertions of `B::f` and `E::f` are evaluated; those of `A::f`, `C::f`, and `D::f` are not.

By contrast, Eiffel, D, and Ada implement narrowing of postconditions by AND-ing the postcondition assertions of the final overrider with those of *all* overridden functions. In the scenario above, those languages would also evaluate the postcondition assertions of `A::f`, `C::f`, and `D::f`, which may trigger contract violations even though they are irrelevant to the correctness of the specific call. In C++, such behaviour could break otherwise correct code.

The following sections examine concrete cases where these and other designs produce false positives or false negatives that our proposal avoids.

4.4 Detecting interface misuse

Consider again the `Display` — `XDisplay` inheritance hierarchy from Section 4.2. What happens if we call `Display::post_message` out of contract, but the dynamic type of the object is `XDisplay`?

```
void boo(Display& display) {
  display.post_message("😈");
}

int main() {
  XDisplay xDisplay;
  test(xDisplay);
}
```

In a design where precondition widening is realised by OR-ing the preconditions of the overriding and overridden functions, such as in Eiffel and D, this program compiles, run, and displays the emoji (assuming that both the source and execution character sets are UTF-8). Even with contract checking enabled, it is sufficient for either `Display::post_message` or `XDisplay::post_message` to satisfy its precondition; in this case, the latter does.

However, this is not a desirable outcome for C++, where `boo` and `XDisplay` may be located in completely different, independently maintained components. We should be able to reason about the correctness of each component individually.

The function `boo` cannot know the dynamic type of its parameter, nor whether it supports UTF-8 input; all it knows is that it is calling a virtual function whose interface requires an ASCII string. The call above is therefore an *incorrect use* of the `Display` interface. In another context, the dynamic type of `display` might be a class that *only* supports ASCII — an entirely correct implementation of the interface — and the same call would then fail, potentially catastrophically if contact checking were disabled.

Such misuse of a polymorphic interface should be caught early. Under our proposed design, this call triggers a violation of the caller-facing precondition assertion, correctly diagnosing the bug at the call site.

4.5 No implicit assertion inheritance

Automatically inheriting precondition and postcondition assertions from overridden functions — as done in Eiffel, D, and Ada — does not work well for C++.

First, inheriting contract assertions from a base class is conceptually incorrect when inheritance is used solely to obtain dynamic dispatch and not to express behavioural subtyping. This is common in idioms specific to C++ such as type erasure.

Second, even in more conventional object-oriented designs, base and derived classes often reside in different components or libraries. Each must be able to define or modify its own assertions independently without breaking the other component or introducing false positives.

Consider a `Widget` base class in a C++ GUI framework that contains a virtual function with the following plain-language contract:

```
class Widget {
public:
  // Preconditions: child points to a Widget; child->isValid() is true;
  // zOrder is between -32768 and +32767.
  virtual void addChildWidget (Widget* child, int zOrder) = 0;
};
```

Because the framework predates C++26, this contract is not expressed through contract assertions. In a downstream codebase that cannot modify the code of this framework, a hierarchy of custom widget types, all derived from the `Widget` class, are used to build a GUI. For these custom widget types, we may want to introduce precondition assertions to check the requirements imposed by the framework:

```
class CustomWidget : public Widget {
public:
  void addChildWidget(Widget* child, int zOrder) override
    pre (child && child->isValid())
    pre (zOrder >= -32768 && zOrder <= 32767) {
      // custom child/parent widget tracking here...
      Widget::addChildWidget(child, zOrder);
    }
};
```

In a language where precondition assertions are OR-ed with those of overridden functions, these assertions could not be made to work. In Ada, the analogous code would not compile; in Eiffel and D, it would compile but the precondition assertions would never have an effect, since they are implicitly OR-ed with `true`.

By contrast, in our proposal, the above code compiles and behaves as intended: the precondition assertions of `CustomWidget` would be evaluated whenever that class — or any class derived from it — is used.

Consider next the owner of the GUI framework, who now decides to add precondition assertions to the base class itself:

```
class Widget {
public:
  virtual void addChildWidget(Widget* child, int zOrder) override
    pre (child && child->isValid())
    pre (zOrder >= -32768 && zOrder <= 32767);
};
```

What happens now when a client of the GUI framework has widened the preconditions in their own implementation of `Widget` — which is valid and consistent with the substitution principle?

```
class SuperWidget : public Widget {
public:
  // this version can handle any values for zOrder!
  void addChildWidget(Widget* child, int zOrder) override;
};
```

If the base class' assertions were implicitly inherited by all derived classes, as in Eiffel, D, Ada, and many earlier C++ proposals, then any code using `SuperWidget` directly with a `zOrder` larger than 32767 would now fail at runtime with a contract violation, *even though it is correct*.

The owner of `SuperWidget` could attempt to resolve this by explicitly adding wider contract assertions to override the inherited ones, or — as suggested during informal discussions of this design space in WG21 — by using a compiler flag that warns when implicit inheritance occurs. Both approaches, however, require awareness of the issue, which would most likely arise only after a spurious contract violation has already caused the program to fail at runtime, possibly in production. Such false positives are unacceptable.

This issue exposes a fundamental flaw in Bertrand Meyer's original design: it assumes that contract assertions can express the entire plain-language contract across the whole program, while in reality they will only ever express a subset of it, and only in some components. Our proposed C++ design accounts for this fundamental limitation, ensures that introducing contract assertions cannot cause remote breakage of correct code, and enables the user to adopt precondition and postcondition assertions on virtual functions *incrementally*.

4.6 Inheritance and contract-violation handling

In the C++ contract-violation handling model, each evaluation of a contract assertion happens with one of four evaluation semantics: *ignore*, *observe*, *enforce*, or *quick_enforce*. The latter two are *terminating semantics* — if any individual contract assertion evaluated with one of these semantics fails, the program will not continue past the assertion and into user-defined code.

Giving the user the ability to configure their program in this manner is an essential design requirement because C++ is prone to undefined behaviour. If the program is found to be in a possibly corrupted state, executing *any user-defined code* could result in a vulnerability. The need to uphold this requirement has guided the design of C++ contract assertions on several occasions ([P3417R1], [P3819R0]). Adopting the model in Eiffel and D, in which precondition assertions are OR-ed across the inheritance hierarchy, would directly undermine this principle.

Consider a virtual function call where `Base::f()` is the statically chosen function and `Derived::f()` is the final overrider. First, we evaluate the precondition assertions of `Base::f()`. Under Eiffel's model, when a precondition fails at this point, the implementation cannot call the contract-violation handler and/or terminate at this point. Instead, it must proceed to evaluate the precondition assertions of `Derived::f()` and OR the results to determine whether contract-violation occurred. This means that even with a critical assertion, such as a bounds check, execution must continue into user-defined code to complete the OR operation.

Such continuation may be acceptable if the user opted into the *observe* semantic, but cannot be reconciled with the *enforce* or *quick_enforce* semantics. The conclusion is that the "OR-ing preconditions, AND-ing postconditions" approach — originally designed by Bertrand Meyer for a *memory-safe* language with no undefined behaviour — is unsuitable for C++.

Instead, the evaluation sequence for precondition and postcondition assertions on virtual functions proposed here preserves the fundamental guarantee already present in C++26: if any individual contract assertion fails, a contract violation occurs immediately.

More fundamentally, OR-ing two sequences of *contract assertions* is not equivalent to OR-ing the two contracts that those assertions check. As we saw above, contract assertions can only validate a subset of a function's plain-language contract. Therefore, assuming that one side of such a disjunction is satisfied merely because none of the associated assertions failed is logically unsound.

4.7 Breaking substitutability by enforcing it

Bertrand Meyer's original design — followed by Eiffel, D, and to a lesser extent Ada, as well as several early C++ proposals — is founded on the premise that the substitution principle should be *enforced by construction*. In other words, the design attempts to guarantee substitutability through language rules rather than through verifying behavioural correctness.

However, statically enforcing substitutability would require subsumption proofs between arbitrary assertion predicates — an undecidable problem. To make the model tractable, Meyer replaced it with a simpler set of rules: "preconditions may only be widened, postconditions may only be narrowed". This set of rules is enforced by OR-ing precondition assertions and AND-ing postcondition assertions across the inheritance hierarchy, the approach chosen in Eiffel and D.

However, this set of rules rejects cases that are perfectly compatible with the substitution principle. Specifically, the substitution principle places no requirements on a derived class when the base class preconditions are not met, while Bertrand Meyer's design forces a derived class to meet the base class postconditions even if it widens the base class preconditions and is then used outside of those base class preconditions. This happens frequently when an overriding function operates on a larger domain than the overridden one.

Consider the following example. Let `Number` be a type that can represent real or complex numbers, and `Sqrt` a type that computes a real-number square root:

```
struct Sqrt {
  virtual Number compute(const Number& x)
    pre(x.isReal() && x.realPart() >= 0)
    post(r : r.isReal() && r.realPart() >= 0);
};
```

The function `Sqrt::compute` is virtual to allow users to substitute their own implementations of `Sqrt` (say, featuring hardware acceleration or enhanced precision). Using virtual functions here may seem contrived at first, but such code appears in real-world contexts, for example in interpreters for built-in scripting languages.

`Sqrt::compute` requires a nonnegative real number as input, and returns another nonnegative real number as output; this contract is specified with contract assertions. Later on, we might add an implementation that can handle the entire complex plane, and therefore drops these contract assertions:

```
struct ComplexSqrt : public Sqrt {
  Number compute(const Number& x) override;
};
```

When used as a `Sqrt`, the derived class `ComplexSqrt` gives clients the same guarantee they expect from `Sqrt`: if they pass a nonnegative real number, they get a nonnegative real number back. `Sqrt` is therefore fully substitutable by `ComplexSqrt`. However, `ComplexSqrt` is not intended to satisfy that postcondition for *all* inputs. When used directly, it can accept and correctly process negative or complex inputs:

```
int main() {
  Number complexUnit = ComplexSqrt().compute(-1);
}
```

The correct output in this case is *not* a nonnegative real number. Yet, in any design that follows "widen preconditions, narrow postconditions", this program would incorrectly trigger a contract violation because the postcondition from `Sqrt::compute` is not met, even though that postcondition is irrelevant for the correctness of the program. By contrast, our proposal evaluates only the assertions associated with the concrete virtual function call, and thus avoids the false positive.

4.8 Narrowing preconditions, widening postconditions

The substitution principle is a cornerstone of object-oriented programming. It ensures that derived classes remain true to the behaviour and expectations of their base classes, allowing them to be used interchangeably without introducing bugs. This property is particularly valuable for public APIs, libraries, and frameworks where external clients depend on the stability of well-defined contracts.

However, not all C++ designs follow this model. In performance-critical or domain-specific systems, or more generally in situations where the same team controls the entire codebase and is aware of the expected usage patterns, unconditional substitutability may be neither required or expected. Derived classes may legitimately *narrow preconditions* or *widen postconditions*, with substitutability established dynamically at runtime rather than statically through the type system.

A common example arises when a derived class requires additional setup or resource acquisition (database connections, threads, etc.). Consider a base class for rendering images:

```
class Image {
public:
  virtual void render() const;
};
```

A derived class may provide GPU-accelerated rendering, which is more efficient but requires explicit initialisation before it can be used:

```
class GPUImage : public Image {
  bool readyToRender = false;
public:
  bool prepare() {
    // upload data to GPU, handle errors...
    readyToRender = true;
    return readyToRender;
  }

  void render() const override
    pre (readyToRender);
};
```

A program that ensures a successful call to `prepare()` always happens before a call to `render()` is *correct*. On the other hand, failure to do so is a bug that should be detectable by contract checking. Our proposal handles this naturally: the precondition assertion of `GPUImage::render()` is evaluated whenever that function is called.

By contrast, designs that treat substitutability as a static property — as in Eiffel, D, Ada, and most earlier C++ proposals — cannot accommodate such usage patterns, even though they are both correct and common in real-world C++ systems (see [P3600R0] for more examples). If C++ contract assertions fail to support them, developers in these domains are unlikely to refactor their code to follow the canonical object-oriented paradigm that requires unconditional substitutability — instead, they will simply not adopt contract assertions.

4.9 Multiple inheritance

Unlike all previous C++ proposals for precondition and postcondition assertions on virtual functions, this proposal fully supports *multiple inheritance*, including cases where base classes specify different precondition and postcondition assertions.
Consider a function-type hierarchy that might appear in a simple expression interpreter. All types in the hierarchy derive from a common abstract base class:

```
struct Function {
  virtual Value compute(const std::vector<Value>& arguments);
};
```

For certain situations, a unary or binary function might be required. These can be modelled as subclasses of `Function` that introduce preconditions on their arguments:

```
struct UnaryFunction : Function {
  Value compute(const std::vector<Value>& arguments) override
    pre(arguments.size() == 1);
};

struct BinaryFunction : Function {
  Value compute(const std::vector<Value>& arguments) override
    pre(arguments.size() == 2);
};
```

Now, we may want to introduce variadic functions that can be used as both unary and binary functions:

```
struct VariadicFunction : public UnaryFunction, public BinaryFunction {
  Value compute(const std::vector<Value>& arguments) override
    /* no preconditions */;
};
```

Code that uses a `UnaryFunction` must pass exactly one argument; code that uses a BinaryFunction must pass exactly two. A `VariadicFunction` satisfies both interfaces and can be safely substituted for either. Under the proposed caller-facing/callee-facing assertions model, all of these cases behave correctly and predictably.

As another example, consider two classes with disjoint contracts that constrain the input domain and the output range differently:

```
struct EvenComputer {
  virtual int compute(int x)
    pre(isEven(x))
    post(r : isEven(r));
};
struct OddComputer {
  virtual int compute(int x)
    pre(isOdd(x))
    post(r : isOdd(r));
};
```

Now consider the following function, which inherits from both and satisfies the contract of both:

```
struct Identity : EvenComputer, OddComputer {
  int compute(int x) override { return x; }
};
```

Identity is substitutable for both `EvenComputer` and `OddComputer`. It should be well-formed and operate without contract violations when used correctly through either interface (or directly). Our proposal achieves this by evaluating the appropriate caller-facing and callee-facing assertions for each call. By contrast, designs based on "widen preconditions, narrow postconditions" — as in Eiffel, D, and Ada — cannot represent such cases correctly.

4.10 Interoperability with other language features

Our proposed caller-facing/callee-facing assertions model generalises cleanly to all existing C++ language features that involve virtual functions. No special rules or corner cases must be carved out — the same simple rules apply universally.

We already encountered assertions on *pure virtual functions* and an example of their use in a *diamond-shaped inheritance hierarchy* in previous sections. Precondition and postcondition assertions on functions in *virtual base classes* are also allowed and behave according to the same rules — the caller-facing assertions (those of the statically chosen function) and the callee-facing assertions (those of the final overrider) will be evaluated in each case.

Virtual function calls through a *pointer-to-member-function* also fit seamlessly into this model. The caller is dereferencing the pointer; since contract assertions cannot be associated with pointers, the caller-facing assertions are simply an empty set. The callee-facing assertions are those of the pointed-to function. The resulting behaviour is identical to the existing behaviour of pointers to *non-virtual* functions in C++26.

5 Extensibility

While this proposal already covers a broad range use cases and provides the correct defaults, more fine-grained control can be introduced through *opt-in syntax* to support even more use cases. Such extensions allow advanced users to model even more specific behaviours while preserving the simplicity and predictability of the base model.

We do not propose these extensions at this time, as some finer details and the exact syntax remain open. All of them can be added incrementally at a later point in time with no breaking changes.

We discussed in Section 4.5 that implicit inheritance of assertions from overridden functions is not a viable *default* for C++, as it can cause remote code breakage and false positives. However, there are cases where controlled inheritance of assertions can be useful.
With the base proposal, if an overriding function wishes to have the same sequence of precondition and postcondition assertions as the overridden function, and have that sequence evaluated even when called directly, it needs to repeat the sequence. This can be impractical due to verbosity, or even impossible if a predicate in the overridden function refers to a private member. Worse, identical expressions may acquire different meanings in derived classes due to different entities being found by name lookup.

Instead of repeating the sequence on the overriding function, an opt-in syntax could allow explicit inheritance. The exact syntax remains open for discussion, but it could take the following form:

```
struct Derived : Base {
  void f(int i, int j) override
    pre(Base::f)
};
```

Here, `pre(Base::f)` explicitly requests that the preconditions of `Base::f` be inherited. Because a function may override multiple virtual functions simultaneously, the function to inherit assertions from must be named explicitly.

Further specification is needed to ensure that the inherited assertions are applied and interpreted correctly: name lookup needs to happen in the context of the base class, the type of `*this` needs to be that of the base, etc. Notably, all previous C++ proposals that featured assertion inheritance — including C++20 Contracts [P0542R5] and their implementation in GCC — failed to do this correctly.

The owner of a base class may also wish to specify that overriding functions *must* inherit its assertions. As an example, consider `QIODevice`, the base interface class of all I/O devices in the Qt framework.[4] It defines a number of virtual functions such as `readData` that take a raw pointer to read bytes from. A future version of Qt might add a precondition assertion to check that the passed-in pointer is not null:

```
class QIODevice {
public:
  virtual qint64 readData(char* data, qint64 maxSize)
    pre (data != nullptr) = 0;
};
```

When a user inherits from `QIODevice` and overrides `readData`, this assertion is likely to be useful to them, but adding it on the overriding function is easy to forget; it would be beneficial if that could be made harder. As we saw above, inheriting the precondition assertion cannot be the silent default for all classes — for successful adoption of any tool designed to diagnose bugs, it is far more important to avoid false positives than it is to avoid false negatives — but could be an opt-in along the lines of:

```
virtual qint64 readData(char* data, qint64 maxSize)
  pre inherit (data != nullptr) = 0;
```

Such an add-on would have to answer a few design questions: would overriding functions now inherit the assertion implicitly, or would still have to do so explicitly or otherwise be ill-formed? If inheritance is implicit, could the overriding function opt out (perhaps because it can easily handle null pointers), and if so, how?

[4] Documentation available at https://doc.qt.io/qt-6/qiodevice.html

Note that the desired effect can already be accomplished today without such an add-on by using the non-virtual interface pattern.[5] However, this approach requires refactoring the base class function definition, which may not always be possible or practical.

Further, there are use cases where it could be useful to define precondition and postcondition assertions to be evaluated only when caller-facing or only when callee-facing. As we saw above, the only reasonable default is *both*, but the other options can be enabled via opt-in syntax. For example, consider a base class for an allocator:

```
struct Allocator {
  void* allocate(size_t size)
    pre (size < maxSize) = 0;
};
```

Now, we might want to introduce a dummy implementation in this base class — perhaps for testing or mocking purposes, or as a fallback that overriders can invoke via a directly qualified call — that never allocates anything and always returns a null pointer.

This dummy implementation can handle any value for `size` just fine — the precondition applies only to the interface, not to the implementation. On the other hand, it has a postcondition — the return value is always null — which only applies to the implementation, and not to the interface. We can express such a contract with opt-in syntax as follows:

```
struct Allocator {
  void* allocate(size_t size)
    pre interface (size < maxSize)
    pre implementation (r: r == nullptr);
};
```

In the evaluation sequence for precondition and postcondition assertions on virtual functions, assertions marked with `interface` will be skipped when they are callee-facing, and assertions marked with `implementation` will be skipped when they are caller-facing.

Taken together, the set of extensions described in this section provides *all* of Ada's functionality in this area and more broadly *all* currently known use cases for precondition and postcondition assertions on virtual functions (see [P3600R0]).

[5] Replace the public virtual function in the base class with a public non-virtual function; place the precondition and postcondition assertions on that function; then make that function internally invoke a private virtual function that derived classes can override.

6 Proposed wording

The proposed wording is relative to the current C++ working draft [N5014]. It is a rebased version of the wording for `pre` and `post` on virtual functions which was present in [P2900R13], approved by EWG, reviewed by CWG and then — at EWG's request — taken out again in [P2900R14] before adoption of the latter into the C++ working draft. We also added a code example, which was missing in [P2900R13].

Modify [intro.execution] paragraph 11 as follows:

> When invoking a function *f* (whether or not the function is inline), every argument expression and the postfix expression designating *f* are sequenced before every precondition assertion of ~~*f* ([dcl.contract.func])~~the function call ([expr.call]), which in turn are sequenced before every expression or statement in the body of *f*, which in turn are sequenced before every postcondition assertion of ~~*f*~~the function call.

Add new paragraphs to [expr.call] after paragraph 5:

> The precondition assertions of a function call are, in order,
>
> - if the postfix expression is a (possibly implicit) class member access expression and the call is a virtual function call, the precondition assertions ([dcl.contract.func]) of the statically chosen function, then
> - the precondition assertions of the function that is being called.
>
> The postcondition assertions of a function call are, in order,
>
> - the postcondition assertions of the function that is being called, then
> - if the postfix expression is a (possibly implicit) class member access expression and the call is a virtual function call, the postcondition assertions of the statically chosen function.
>
> [Example:

```
struct X1 { virtual void f()          pre(a) post(b); };
struct X2 { virtual void f()          pre(c) post(d); };
struct Y : X1     { void f() override pre(e) post(f); };
struct Z : Y, X2 { void f() override pre(g) post(h); };

void t() {
  Z z;
  z.f();                           // asserts g, h
  static_cast<Y*>(&z)->foo();      // asserts e, g, h, f

  X1& x1ref = z;
  X2& x2ref = z;
  x1ref.f();                       // asserts a, g, h, b
  x2ref.f();                       // asserts c, g, h, d
  x1ref.X1::f();                   // asserts a, b
```

```
    void (X1::*pmf)() = &X1::f;
    (x1ref.*pmf)();              // asserts g, h
  }
```

— end example]

Modify [expr.call], paragraph 6 as follows:

> When a function is called, each parameter ([dcl.fct]) is initialized ([dcl.init], [class.copy.ctor]) with its corresponding argument, and each precondition assertion of the function call is evaluated ([dcl.contract.func]). If the function is an explicit object member function and there is an implied object argument ([over.call.func]), the list of provided arguments is preceded by the implied object argument for the purposes of this correspondence. If there is no corresponding argument, the default argument for the parameter is used.

Modify [expr.call], paragraph 7 as follows:

> The *postfix-expression* is sequenced before each *expression* in the *expression-list* and any default argument. The initialization of a parameter or, if the implementation introduces any temporary objects to hold the values of function parameters ([class.temporary]), the initialization of those temporaries, including every associated value computation and side effect, is indeterminately sequenced with respect to that of any other parameter. These evaluations are sequenced before the evaluation of the precondition assertions of the function call, which are evaluated in sequence ([dcl.contract.func]). For any temporaries introduced to hold the values of function parameters, the initialization of the parameter objects from those temporaries is indeterminately sequenced with respect to the evaluation of each precondition assertion.

Modify [expr.call], paragraph 9 as follows:

> When the called function exits normally ([stmt.return], [expr.await]), all postcondition assertions of the function call are evaluated in sequence ([dcl.contract.func]). If the implementation introduces any temporary objects to hold the result value as specified in [class.temporary], the evaluation of each postcondition assertion is indeterminately sequenced with respect to the initialization of any of those temporaries or the result object. These evaluations, in turn, are sequenced before the destruction of any function parameters.

Modify [dcl.contract.func] paragraph 6 as follows:

> A ~~virtual function ([class.virtual]), a~~ deleted function ([dcl.fct.def.delete])~~,~~ or a function defaulted on its first declaration ([dcl.fct.def.default]) shall not have a *function-contract-specifier-seq*.

Acknowledgements

Thanks to Gašper Ažman, John Lakos, Nina Ranns, Lisa Lippincott, and Herb Sutter for the many conversations that led to this revision of the paper. Thanks to Sam Saariste and Oliver Rosten for reviewing this revision and providing valuable feedback. Thanks to Ville Voutilainen, Ran Regev, Tom Honermann, and Andrzej Krzemieński for reviewing previous revisions of this paper and providing valuable feedback. Thanks to Lisa Lippincott for contributing the very helpful caller/callee diagram.

Bibliography

[N4110] J. Daniel Garcia: "Exploring the design space of contract specifications for C++". 2014-07-06

[N4415] Gabriel Dos Reis, J. Daniel García, Francesco Logozzo, Manuel Fähndrich, and Shuvendu Lahirii: "Simple Contracts for C++". 2015-04-12

[N5014] Thomas Köppe: "Working Draft, Programming Languages — C++". 2025-08-05

[OOSC2] Bertrand Meyer: "Object-Oriented Software Construction", 2nd edition. Prentice Hall, 1997

[P0147R0] Lawrence Crowl: "The Use and Implementation of Contracts". 2015-11-08

[P0247R0] Nathan Myers: "Criteria for Contract Support". 2016-02-12

[P0287R0] Gabriel Dos Reis, J. Daniel García, Francesco Logozzo, Manuel Fähndrich, and Shuvendu Lahiri: "Simple Contracts for C++". 2016-02-15

[P0380R0] Gabriel Dos Reis, J. Daniel García, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup: "A Contract Design". 2016-05-28

[P0542R5] Gabriel Dos Reis, J. Daniel García, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup: "Support for contract based programming in C++". 2018-06-08

[P2388R4] Andrzej Krzemieński and Gašper Ažman: Minimum Contract Support: either *No_eval* or *Eval_and_abort*. 2021-11-15

[P2521R5] Andrzej Krzemieński, Gašper Ažman, Joshua Berne, Bronek Kozicki, Ryan McDougall, and Caleb Sunstrum: "Contract support — Record of SG21 consensus". 2023-08-15

[P2885R3] Timur Doumler, Gašper Ažman, Joshua Berne, Andrzej Krzemieński, Ville Voutilainen, and Tom Honermann: "Requirements for a Contracts syntax". 2023-10–05

[P2899R1] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++ — Rationale". 2025-03-14

[P2900R13] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++". 2025-01-13

[P2900R14] Joshua Berne, Timur Doumler, and Andrzej Krzemieński: "Contracts for C++". 2025-02-13

[P2932R3] Joshua Berne: "A Principled Approach to Open Design Questions for Contracts". 2024-01-15

[P2954R0] Ville Voutilainen: "Contracts and virtual functions for the Contracts MVP". 2023-08-03

[P2961R2] Timur Doumler and Jens Maurer: "A natural syntax for Contracts". 2023-11-08

[P3097R0] Timur Doumler, Joshua Berne, and Gašper Ažman: "Contracts for C++: Support for Virtual Functions". 2024-04-12

[P3173R0] Gabriel Dos Reis: "P2900R6 May Be Minimal, but It Is Not Viable". 2024-02-15

[P3327R0] Timur Doumler: "Contract assertions on function pointers". 2024-10-16

[P3417R1] Gašper Ažman and Timur Doumler: "Handling exceptions thrown from contract predicates". 2025-02-27

[P3506R0] Gabriel Dos Reis: "P2900 Is Still not Ready for C++26". 2024-11-19

[P3573R0] Michael Hava, J. Daniel Garcia Sanchez, Ran Regev, Gabriel Dos Reis, John Spicer, Bjarne Stroustrup, J.C. van Winkel, David Vandevoorde, Ville Voutilainen: "Contracts concerns". 2025-01-12

[P3600R0] Timur Doumler: "Contract assertions on virtual functions: a principled design approach". 2025-12-31

[P3819R0] Peter Bindels, Timur Doumler, Joshua Berne, Eric Fiselier, and Iain Sandoe: "Remove `evaluation_exception()` from contract-violation handling for C++26". 2025-09-02