Contracts for C++: Support for Virtual Functions

Document #: P3097R0
Date: 2024-04-12
Project: Programming Language C++
Audience: SG21 (Contracts)
Reply-to: Timur Doumler <papers@timur.audio>
Joshua Berne <jberne4@bloomberg.net>
Gašper Ažman <gasper.azman@gmail.com>

Abstract

We propose to add support for precondition and postcondition assertions on virtual functions to Contracts for C++ ([P2900R6]). Instead of inheriting preconditions and postcondition assertions from an overridden function, we invoke the preconditions and postcondition assertions of both the static and dynamic type when doing virtual dispatch. Our proposed solution mitigates the shortcomings of previous Contracts designs for C++ and enables a broad range of use cases including meaningful support for multiple inheritance.

Contents

1 Motivation

Runtime polymorphism is a fundamentally important part of the C++ language. Many C++ libraries and APIs are designed around virtual functions and runtime polymorphism; it is one of the major styles of writing C++.

In the two decades of attempting to standardise a Contracts facility for C++, it has been recognised as an important design principle that such a Contracts facility must be well-integrated with runtime polymorphism ([N4110]). Yet our current Contracts MVP ([P2900R6]), which is targeting C++26, does not offer any such integration and instead makes contract annotations on virtual function declarations ill-formed. This choice is consistent with the [P2900R6] design principle to "choose ill-formed to enable flexible evolution". However, considering the widespread use of runtime polymorphism in the C++ ecosystem, the resulting design hole has the potential to significantly hamper the adoption of Contracts.

2 Prior Art

2.1 Overview

There are several programming languages that support runtime polymorphism as well as contract assertions as a core language feature, and integrate the two, such as Eiffel, D, and Ada. In C++, prior to [P2900R5] which is the revision of the Contracts MVP proposal that removed support for virtual functions, *all* proposals to standardise a Contracts facility that allowed placing precondition and postcondition assertions on function declarations also supported virtual functions, recognising the importance of integration between Contracts and runtime polymorphism.

For an object to guarantee it is usable wherever another instance of that polymorphic type is usable, it should provide a contract that is universally compatible with the base class contract. In particular, it must have preconditions that are the same or wider than the base class preconditions and, in the cases where the base class preconditions have been met, it must have thesame or narrower postconditions. This principle is often shortened to the inaccurate but easy to remember mantra of "widen preconditions and narrow postconditions" — a form of covariance and contravariance on the contract conditions as you travel up and down a class hierarchy.

If we examine the existing and proposed designs for contract assertions on virtual functions, we find that they almost universally either enforce compliance with this particular form of substitutability — with varying degrees of flexibility — or completely ignore the virtual nature of a function. In particular, we can distinguish five different designs:

A Automatically inherit precondition and postcondition assertions from the overridden function; allow the overriding function to specify their own; when checking contract assertions, enforce the "preconditions must be wider, postconditions must be narrower" rule (typically by OR-ing the precondition assertions and AND-ing the postcondition assertions of the overriding function and those of all other functions in the given inheritance chain).

B Automatically inherit precondition and postcondition assertions from the overridden function; optionally, allow to repeat the same sequence of precondition and postcondition assertions on the overriding function, but never alter it through a class hierarchy

C Require an overridden function to specify the same sequence of precondition or postcondition assertions as the overriding function. This is essentially the same as B but with overriding functions needing to explicitly restate the precondition and postcondition assertions.

D Ignore the virtual nature of a function. Precondition and postcondition assertions get compiled into the implementation of the function and are only checked if that implmentation gets called.

E Do not allow precondition or postcondition assertions on virtual functions at all. This is effectively the same as D, except that we have to spell the precondition and postcondition assertions with assertion statements inside the body of the function.

2.2 Eiffel

In the Eiffel programming language[1], precondition and postcondition assertions are spelled with `require` and `ensure` clauses. Overriding functions ("redeclared versions of a routine") automatically inherit the precondition and postcondition assertions of the overridden function. They are permitted to specify additional precondition and postcondition assertions, which are spelled differently from those in a non-overriding function: `require else` and `ensure then`, respectively.

When checking contract assertions, a `require else` clause will be OR-ed with the original `requires` clause, and a `ensure then` clause will be AND-ed with the original `ensure` clause, enforcing the rule that the precondition of the derived class must not be narrower and the postcondition of the derived class must not be wider than that of the base class. Eiffel therefore implements a variant of Design A.

2.3 D

The D programming language[2] has semantics similar to Eiffel, except that precondition and postcondition assertions in the derived class do not have a special syntax; they are always spelled with `in { ... }` and `out { ... }` clauses. The `in { ... }` clause is OR-ed with that of the base class, while the `out { ... }` clause is AND-ed with that of the base class. D is therefore another instance of Design A.

2.4 Ada

The Ada programming language[3] has an idiosyncratic set of semantics for contract assertions on virtual functions. It has two distinct kinds of precondition and postcondition assertions: *specific* and *class-wide*. Class-wide and specific precondition and postcondition assertions can also be combined on the same function.

Specific precondition and postcondition assertions are spelled with `Pre` and `Post`, are checked callee-side (i.e. inlined into the body of the function), and are not inherited. An overridden and overriding function each have their own independent set of precondition and postcondition assertions; only the set that belongs to the function that ends up being called dynamically will be checked. Specific precondition and postcondition assertions in Ada are therefore a variant of Design D. Because they

[1] https://www.eiffel.org/doc/eiffel/ET-_Inheritance
[2] https://dlang.org/spec/function.html#in_out_inheritance
[3] https://docs.adacore.com/live/wave/arm05/html/arm05/RM-6-1-1.html

are compiled into the body of a function, and abstract functions do not have a body, they cannot have specific precondition and postcondition assertions.

Class-wide precondition and postcondition assertions are spelled with `Pre'Class` and `Post'Class` and are function-contract assertions that apply to the overridden function and all its overriders. The terminology "class-wide" may be somewhat confusing to C++ developers because it does *not* mean that the assertions apply to all member functions of a class — Ada does not have member functions — but to all versions of a function that belong to the same virtual function tree. An overriding function can redefine class-wide precondition and postcondition assertions.

Class-wide precondition assertions are checked callee-side (i.e. inlined into the calling code). The precondition assertions visible at the call site are combined with a short-circuiting OR executed in unspecified order; preconditions not visible at the call site (e.g., preconditions on the derived version where the call is made through a base interface) are not checked.

By contrast, class-wide postconditions are checked caller-side (i.e. inlined into the implementation); the postconditions of the overridden function and those of all functions in the chain of overriders up to the one being called will be checked and AND-ed together.

Therefore, in our classification, Ada is a hybrid: class-wide preconditions have semantics similar to design A, except that only preconditions visible at the call site are checked, while class-wide postconditions follow Design A exactly, and specific preconditions and postconditions follow Design D.

2.5 C++

The first Contracts for C++ proposal [N1613] was directly inspired by the D language and therefore proposed Design A. In the next revision of that paper, [N1669], the authors retained Design A for postcondition assertions (which are AND-ed together), but instead adopted Design C for precondition assertions with the rationale that the previous design is "useless in practice since the user cannot take advantage of the weaker precondition by performing a downcast" (a problem that our proposal solves).

Later Contracts for C++ proposals, including C++20 Contracts and the Contracts MVP work that followed it, all used either Design B ([N4415], [P0542R5], [P2388R4]) or Design C ([P0380R0]), neither of which allows an overriding function to modify the contract of the overridden function. This only changed when adopted Design E for the Contracts MVP in [P2900R5].

The intent of Design E was not that virtual functions do not deserve to have precondition and postcondition assertions on them, but to defer support for them until consensus on the correct design can be achieved. It arose as temporary consensus in SG21 because we found that designs A — D do not support all the current uses of virtual functions common in C++ and therefore rejected them (see [P2932R3] section 3.4). More, it was not obvious that any of those solutions would support evolving to any more general solution.

3 Use Cases

3.1 Placing Contract Annotations on Virtual Functions

Consider a class with a virtual function having a precondition:

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

When calling this function out of contract, we expect a contract violation to be detected:

```
void testCar(Car& car) {
  car.drive(90);  // OK
  car.drive(120); // bug: precondition violation
}
```

This is a very basic and perfectly valid use case for a precondition annotation which we expect to be very common once we introduce Contracts to C++. The language should support this use case, yet designs D and E do not allow the user to express this.

3.2 Widening Preconditions

It was first noted in [P0247R0] that inheriting precondition and postcondition assertions from the overridden function (design B), or requiring that precondition and postcondition assertions on the overriding function match the overridden function (design C), also precludes important use cases. In C++, it is routine for overriding functions to widen preconditions and narrow postconditions; doing so does not compromise substitutability of the base class by the derived class. Consider the following derived class, overriding `Car::drive` and specifying a wider precondition:

```
struct FastCar : Car {
  void drive(float speed_mph) override
    pre (speed_mph < 150);
};
```

This is a perfectly reasonable and common thing to do and matches good practice when designing polymorphic classes. A derived class should be able to extend the functionality of the base class and accept arguments that are designed to work with that extended functionality. The Contracts facility should support this use case. Designs B and C do not.

Often, the user will still expect code that uses the `Car` interface to comply with the contract of `Car::drive`, even when the dynamic object used at runtime happens to be a `FastCar`; otherwise, our function might fail should it ever be passed a different dynamic type:

```
int main() {
  FastCar myFastCar;
  testCar(myFastCar);
}

void testCar(Car& car) {
  car.drive(90);  // OK
```

```
    car.drive(120); // this is too fast for a Car!
  }
```

The call `car.drive(120)` is an out-of-contract call because it violates the preconditions of `Car::drive`, which is the interface we use here, and `testCar` will have failed to meet the obligations it is aware of.

At the same time, in C++ we are not bound to call `FastCar::drive` through the base class interface. Instead, we can use the derived class directly, in which case only the contract of `FastCar` is relevant:

```
  void testFastCar(FastCar& myFastCar) {
    myFastCar.drive(120);   // no problem here!
  }
```

The same is true if we use a qualified call and thus bypass the virtual function call mechanism entirely:

```
  int main() {
    FastCar myFastCar;
    myFastCar.FastCar::drive(120);   // no problem here, either!
  }
```

In both cases above, the interface `Car::drive` is not used and no virtual function call involving the class `Car` is taking place, so the precondition of `Car::drive` is not relevant to the correctness of the program. The call `drive(120)` is therefore correct and should not trigger a contract violation. None of the designs B—E support this use case.

3.3 Narrowing Preconditions

Now let us consider the case where we want to narrow a precondition in a derived class:

```
  struct ElectricCar : Car {
    void drive(float speed_mph) override
      pre (is_charged);

    void charge() { is_charged = true }
  private:
    bool is_charged = false;
  };
```

Narrowing a precondition in a derived class arguably does not conform to orthodox object-oriented design, because it means that not all instances of `ElectricCar` are substitutable for `Car` in all possible cases:

```
  int main() {
    ElectricCar myElectricCar;
    testCar(myElectricCar);
  }

  void testCar(Car& car) {
    car.drive(90);   // precondition violation: forgot to charge!
  }
```

However, there are situations where such a design makes sense — the more specific preconditions can be guaranteed non-locally by ensuring in a wider scope that we use instances of `ElectricCar` that are ready to be used. C++ Contracts should not demand that we guarantee all conditions in only the tightest scope imaginable, as it makes real use cases and design infeasible. Here is an example for a correct program using the function `testCar` with an `ElectricCar`:

```
int main() {
  ElectricCar myElectricCar;
  myElectricCar.charge();
  testCar(myElectricCar);  // everything works now!
}
```

Note that a default-constructed `ElectricCar` is not immediately substitutable for `Car` upon construction, but becomes substitutable for `Car` after an additional setup step specific for that class. The precondition assertion of `ElectricCar::drive` is designed to ensure that this additional setup step is not omitted. Such designs are common in C++ — and more importantly common in any design that attempts to model the real world — and the precondition assertion is doing exactly what it is supposed to do — diagnosing incorrect use.

In order for this design to work correctly, the precondition of `ElectricCar::drive` needs to be checked every time `ElectricCar::drive` is called, even if the call happens through the base class interface `Car::drive`, otherwise we have failed to diagnose the bug on incorrect use. Our design for C++ Contracts must therefore enforce this check. None of the designs A — E support this use case.

3.4 Narrowing Postconditions

Subclasses that have compatible contracts with their base class can often provide additional guarantees that the base class does not require of all potential subclasses. Failing to meet these additional guarantees is clearly a bug in the subclass that should be detected, even when the direct caller does not directly depend on that guarantee because it is using the base class interface.

Consider an abstract class that is used to access sequences of values through a single virtual function that has a wide contract:

```
template <typename T>
struct Generator
{
  virtual T next();
};
```

Many different derived classes might be implemented that produce values in different ways. Consider, for example, a simple generator that always produces the same value with each call to `next`:

```
template <typename T>
struct ConstantGenerator
{
  const T &getValue() const;    // access the value that will be produced
  T next() override
    post( r : r == getValue() );
};
```

Any reader, and any code directly using a `ConstantGenerator`, can benefit from knowing this postcondition of `ConstantValue::compute`. This postcondition, however, is not the same as the (empty) set of postconditions on overridden function `Generator::next`, and is therefore, despite being fully compatible with the substitution requirement, not supported by designs B — E.

3.5 Widening Postconditions Outside Base Class Contract

Substitutability, though often simplified to "widening preconditions, narrowing postconditions", actually does not need to put any requirements on a function when the base class preconditions are not met; this is a common misrepresentation of the Liskov substitution principle.

Consider a function object which computes a real-number square root and which can be applied to a generic `Number` type that represents real or complex numbers:

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

This implementation of square root might have a number of useful properties, such as being implemented very quickly with hardware instructions and guaranteeing specific tolerances on the accuracy of the result.

On the other hand, we might later on invest in an even better implementation that handles the entire complex plane (and possibly even use the more optimized real-numbers only `Sqrt` in the implementation of this function):

```
struct ComplexSqrt : public Sqrt {
  Number compute(const Number& x) override
    // wide contract
    post( r : (x.isReal() && x.realPart() >= 0)
              ? (r.isReal() && r.realPart() >= 0)
              : true );
};
```

When used as a `Sqrt`, this function will still give clients the guarantees they expect from any implementation of `Sqrt` - namely that they must give it a nonnegative real number and they will get back a nonnegative real number. Consider, for example, this function that itself has a wide contract and uses an instance of `Sqrt` to compute the fourth root of a given `Number`:

```
Number quadroot(Number value, const Sqrt& sqrt) {
  if (!value.isReal() || value.realPart() < 0) {
    throw std::domain_error("Must pass a nonnegative real number");
  }
  Value v = sqrt.compute( {value} );  // v is a nonnegative real number
  return sqrt.compute( {v} );         // preconditions satisfied
}
```

Client code such as this last line of `quadroot`, which invokes a virtual function through a base class interface, invariably depends on the postconditions of that base class interface — and thus the postconditions of the base class should be checked when calling an object of derived class through a base class interface.

Naturally, the implementation `ComplexSqrt` does not — and cannot — guarantee that it will return a nonnegative real number in all cases, in particular when the preconditions of the base class are not met. When handed any negative real number or imaginary number it will still successfully give a complex answer that satisfies the derived function's contract, thus violating the postcondition of the version in the base class `Sqrt`. This implementation is, however, substitutable for `Sqrt` in all cases since it does satisfy the base class postcondition whenever handed a nonnegative real number — i.e. whenever the base class preconditions are satisfied. Once the base class preconditions are violated, all guarantees related to the base class are no longer relevant. Despite being fully compatible with the substitution requirement, this use case is not supported by any of the designs A — E.

3.6 Widening Postconditions

For postconditions, there is a dual to the case where we narrow preconditions (Section 3.3) — namely, a type that will not meet the base class postconditions in general (and is thus not fully substitutable for the base class) but will meet the base class preconditions in specific cases.

Consider, for example, a test function that can be set to expect any specified value as an argument for and return any specified value from the next call to compute:

```
template <typename Base>
void TestFunction : public Base {
  void setInput(const std::vector<Value>& expectedArguments);
  const std::vector<Value>& getInput() const;

  void setOutput(const Value& value);
  Value getOutput() const;

  Value compute(const std::vector<Value>& arguments) override
    pre( arguments == getInput() )
    post( r : r == getOutput() );
};
```

Now, an algorithm that requires a `Sqrt` implementation might still run correctly to completion, likely during testing, when used with the above function:

```
Value sumSqrts(const std::vector<Value>& values, const Sqrt& sqrtFunction);

void testSumSqrts() {
  // verify that sumSqrts works correctly to sum a vector of 4s:
  std::vector<Value> values = { 4, 4, 4, 4 };
  TestFunction<Sqrt> testSqrt;
  testSqrt.setInput( {4} );
  testSqrt.setOutput( {2} );
  ASSERT( 8 == sumSqrts(values, testSqrt) );
}
```

Such limited use implementations of functions can be completely correct when you have additional information about how the polymorphic object will be put to use in particular circumstances, even if again the generalized postconditions would not be suitable for other uses. However, this use case is not supported by any of the designs A — E.

3.7 Multiple Inheritance

Finally, another problem with existing designs is that they generally do not work with multiple inheritance.

For instance, it is completely reasonable to inherit from multiple base classes which have mutually exclusive preconditions on a function, and provide a wider-contract implementation that satisfies both domains. Consider a function-type hierarchy that might be used, for example, when writing a simple expression interpreter, where all types in the hierarchy extend 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 and we could choose to model this as subclass of `Function`:

```
struct UnaryFunction {
  Value compute(const std::vector<Value>& arguments) override
    pre(arguments.size() == 1);
};
struct BinaryFunction {
  Value compute(const std::vector<Value>& arguments) override
    pre(arguments.size() == 2);
};
```

Now, of course, there are 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 */;
};
```

Any code that uses a `UnaryFunction` is obliged to pass a single-element list of arguments to that code. Similarly, two arguments are necessary for anything that uses `BinaryFunction`. Functions that make use of a `UnaryFunction`, however, will work completely correctly when handed a variadic function that could take additional arguments when used differently. Designs like B and C that require overrides to match those of base classes fail completely to allow reasonable class hierarchies such as this.

Design A, which would OR the preconditions of all three functions involved, does allow the case above, but breaks other useful designs involving multiple inheritance. For example, a virtual function might override two functions that have not only mutually exclusive preconditions, but also mutually exclusive postconditions, and yet be fully substitutable for either. Consider a function that requires an even number as input and guarantees an even number as output, and another that operates in the same fashion on odd numbers:

```
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; }
}
```

Despite fully satisfying the substitutability requirement, this function is not supported by any of the designs A — E.

4 Discussion

4.1 Why Minimal Solutions (Designs C — E) Do Not Work

Design E, not allowing precondition and postcondition assertions on virtual functions at all, is a temporary step in fixing a broken design, but we do not consider it viable as the final state of Contracts support in C++ — even though an initial release of Contracts could be viably made without that support. Vast amounts of C++ code are written using virtual functions and runtime polymorphism, which is one of the major styles of writing C++; Contracts should support it.

Design D, only evaluating the function-contract assertions of the implementation, is not viable even as an intermediate step, because it would bake semantics into the language that would preclude any meaningful integration of Contracts with virtual functions at a later stage. If we choose to not evaluate certain existing contract assertions now, there will be no viable way to start evaluating them at a later date with a change to the C++ Standard — the possible runtime breakage of programs written with different assumptions will be insurmountable.

Design C, inheriting contracts but requiring that they be repeated, could be considered as an intermediate step, as this design leaves open several possible directions for future evolution. However, any actual use of function-contract assertions on virtual functions with Design C would require so much typing (all function-contract assertions have always be repeated on every overriding function) and would enable so few additional use cases compared to Design E that the benefit-cost ratio looks rather unfavorable.

4.2 Why Implicitly Inheriting Contracts (Design B) Is a Poor Solution

The success of adoption of a contracts facility will depend, significantly, on the ability to add contract assertions without compromising the correctness or well-formedness of functions not clearly related to the newly added contract assertion. In particularly, adding a function-contract assertion to a base class virtual function must not compromise code which uses a derived class override of that function and makes no mention of the base class. Doing so would violate one of the design principles of [P2900R6] (described in more detail in [P2932R3]) that any given contract assertion should check the compliance with only a specific function contract, and not intrude on other tangentially related function invocations.

Consider, for example, a base class virtual function that has a narrow contract, and a derived class that chooses to provide a wide contract and throw exceptions on the out-of-range inputs:

```
class Base {
  virtual int compute(int x) pre(x >= 0 && x <= 1000);
};

class Derived {
  int compute(int x) override {
    if (x < 0 || x > 1000) { throw range_error(); }
    // ...
  }
};
```

Users of `Derived` depend upon that function to change inputs outside the supported range into exceptions — likely handling that exception in an appropriate fashion as a known, and valid, alternative control flow when using `Derived::compute`. A reader of `Derived` will never even see the function-contract assertions on `Base::compute` to be aware that they might have an impact on their use of `Derived::compute`.

Consider also that `Derived::compute`, without the `override` keyword indicating that it is an intentional override, may not even have been written to intentionally override `Base::compute` and the inheritance of `Base` might be for an unrelated purpose not involving that function. If `Base::compute` is newly added then surely all users of `Derived::compute` will be unprepared and unhappy with the function they previously used without virtual dispatch suddenly having a narrow contract.

These considerations leads to two consequences for how we specify contract assertions on virtual functions:

1. The addition of function-contract assertions on a base class member function must not require explicit opt-in or opt-out of inheritance of those assertions by a derived class. Not having a reasonable default behavior for existing functions would render all overrides, even those in remote libraries completely outside of the control of the author of the base class, ill-formed without a corresponding global addition of the needed syntax to explicitly choose a behavior.

2. When directly invoking an override that has no explicit function-contract assertions, no function-contract assertions should take effect. If this were to happen, the addition of function-contract assertions to the base class would result in breaking code that worked exactly and correctly as originally designed.

4.3 Why Classic Design-By-Contract (Design A) Is a Poor Solution

Design A builds upon design B and therefore suffers from the same issues related to implicitly inheriting contracts. Unlike design B, design A *does* allow overriding functions to specify their own function-contract assertions, but it enforces the classic oversimplified rule for substitutability "preconditions can only be widened and postconditions can only be narrowed", typically via "OR-ing preconditions, AND-ing postconditions" (D, Eiffel, partially Ada). As we have seen in the discussion of use cases 3.5 and 3.7, this rule actually fails to support several cases in which the derived class is fully substitutable for the base class.

One might contemplate enforcing substitutability via subsumption proofs applied to the predicates of the function-contract assertions of an overriding function. However, such proofs would involve reasoning at compile time about arbitrarily complex expressions with arbitrary runtime behavior, so it does not seem like a feasible route for C++ in practice.

With the Contracts model as proposed in [P2900R6], it is also important to note that "OR-ing preconditions" is not even well defined. Contract assertions are in general not usable for control flow since they may be evaluated with any contract semantic, including the *ignore* semantic. There is no way to know after evaluating the precondition assertions of a base class function whether they were violated or not, as any subset may have simply not been checked. Therefore, there is no way to know after evaluating one set of precondition assertions whether the other set can (or even should) be evaluated.

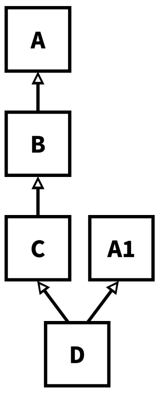
Regardless of how general substitutability is or is not enforced, the use cases presented above indicate that any such design will invariably prevent completely valid use cases and too broadly invalidate software that needs to be written and checked for actual correctness and not adherence to arbitrary abstract concepts. Substitutability is important for "good object-oriented design", but virtual functions are also used in many other C++ techniques such as type-erasure, the pImpl idiom, to provide a stable ABI for a generic shared library interface (e.g. COM or CORBA), implementation (as opposed to "interface") inheritance, (possibly multiple) inheritance to satisfy a callback interface, and so on. In many of these cases, substitutability has little to no relevance, as virtual calls happen through interfaces in the middle or at the leaves of the inheritance hierarchy. In other cases, substitutability is a dynamic property: a derived class can fail to be substitutable for the base class upon construction, but become substitutable after some additional initialization step; contract assertions can be extremely useful for ensuring that this initialization step actually happens and avoiding bugs with such designs. In short, enforcing substitutability prevents the user from adding Contracts to code that would benefit most from them.

4.4 Towards a Better Design

In order to move forward towards a better design, we must ask the question which function-contract assertions should be allowed or enforced when defining an override of a virtual function, and which should be evaluated when a function is invoked via virtual dispatch.

For virtual functions, there is a split between the interface seen by a caller and the interface an implementation is striving to implement that is not present for non-virtual functions. The caller knows only the static type of the reference or pointer through which the function is being invoked, while the callee knows precisely the dynamic type of the object on which the function is being invoked.

As an example that allows us to examine all of the relevant cases, consider the following class hierarchy:



Now, let us add a single virtual function to this hierarchy, which is overridden in all derived classes and has distinct precondition and postcondition specifiers in every class:

```
struct A         { virtual void f()  pre(#1) post(#2);  };
struct A1        { virtual void f()  pre(#3) post(#4);  };
struct B : A     { void f() override pre(#5) post(#6);  };
struct C : B     { void f() override pre(#7) post(#8);  };
struct D : C, A1  { void f() override pre(#9) post(#10); };
```

Now, let us consider the case where `D::f` is being invoked through a pointer or reference of type `B`. Note that there are two other base classes — `A` and `A1` — of `B` and `D` respectively. There is also an additional intermediate class (with a corresponding declaration of `f` with distinct function-contract assertions) `C` between `B` and `D`.

First, we will consider the caller, who only knows the type `B` when invoking `f`:

```
void caller(B& b) {
  // #X
  b.f();
  // #Y
}
```

As with any other function invocation, `caller` knows only the declaration of `B::f` and the function-contract assertions (`#5` and `#6`) attached to that declaration. It could consider the function-contract assertions of `A::f`, which `B::f` overrides — but in this case it has a more specific type and might be trying to take advantage of the specific guarantees of `B`'s contract that are compatible with but distinct from those of `A`. In particular, if `caller` depends on the contract of `A` then the code in block `#X` (and leading up to any invocation of caller) must guarantee the preconditions of `B::f`, and the code in block `#Y` can depend on the postconditions of `B::f`. Any violation of the postconditions of `B::f` would inevitably open up a chance of a defect in `#Y`.

Looking at the callee, `D::f`, we actually have much less information about the function invocation:

```
void D::f() {
  // #Z
}
```

The code in block #Z does not even know if virtual dispatch occurred, let alone what static type was known by the caller on this invocation. In fact, the only function-contract assertions this function can be written against are those of `D::f`. The preconditions and postconditions of `A::f`, `A1::f`, `B::f`, and `C::f` are all potentially applicable — but not to all invocations of `D::f`. A violation of the preconditions of `D::f` (#9) will inevitably lead to defects in the block #Z that is written to assume those preconditions, while a violation of the postconditions of `D::f` (#9) may or may not lead to a defect due to assumptions on the call site being broken, depending on the static type that was called.

Now for any given function declaration, when any of the precondition and postcondition assertions of that function are evaluated, they should all be. This is important because the full set of checks on a function declaration are often logically a cohesive whole; but it is even more important because of other impending features: postcondition captures and procedural function interfaces — where the split between precondition and postcondition checking is not as clear-cut and making any decision to treat them differently would end up being arbitrary and unsupportable.

Therefore, for our invocation of `D::f` through a reference to `B`, we should check only the function-contract assertions of `B::f` and `D::f - #5`, #6, #9, and #10. Function-contract assertions from base classes which the caller is aware of but not directly using (#1 and #2), as well as those from base classes of the callee that it does not know are being actively used (#3, #4, #7, and #8) are not evaluated.

5 The Proposal

5.1 No Inheritance of Overridden Contracts

We propose to allow precondition and postcondition specifiers on virtual functions. The precondition and postcondition specifiers of the overridden function are not inherited by the overriding function; instead, the precondition and postcondition specifiers of an overriding function are completely independent from those of the overridden function. This choice enables all of the use cases we elaborated on in the previous section, allowing both widening and narrowing of either preconditions or postconditions in derived classes.

5.2 Sequence of Contract Checks

Whenever a virtual function call happens, function-contract assertions are evaluated as follows:

1. The preconditions of the “interface” function are evaluated
2. The preconditions of the “implementation” function are evaluated
3. The body of the “implementation” function is executed
4. The postconditions of the “implementation” function are evaluated
5. The postconditions of the “interface” function are evaluated

The "interface" function is the one in the statically called type, determined by the type of the pointer or reference through which the virtual function is invoked. The "implementation" function is the one in the actual dynamic type of the object.

When calling a virtual function directly through the interface of the dynamic type, i.e, when the statically called type and the dynamic type of the object coincide, only the preconditions and postconditions of that function will be evaluated. Performing these evaluations only once is allowed under the Contract MVP rules for elision and duplication of contract checks ([P2900R6] sections 3.5.5. and 3.5.6).

When using a qualified call and thus bypassing the virtual function call mechanism entirely, only the preconditions and postconditions of the called function will be evaluated, just like for non-virtual functions.

This proposed sequence of evaluations checks exactly the function-contract assertions that we concluded in the last section would identify defects in the code surrounding and within the function invocation. The preconditions and postconditions of the interface (overridden function) need to be checked because violating its preconditions would constitute an incorrect call of that interface, and violating its postconditions would mean returning from the function to a state that is incorrect when using that interface. The preconditions and postconditions of the implementation (overriding function) need to be checked because violating its preconditions would constitute an incorrect call for that dynamic type, potentially leading to undefined behaviour in the implementation of the overriding function if ignored, and violating its postconditions would indicate a bug inside that implementation. In other words, the precondition and postcondition checks on the interface enforce the requirements and promises regarding the user of that interface, and the precondition and postcondition checks on the implementation enforce the requirements and promises regarding executing that implementation.

This design ensures there are always exactly two sources of contract assertions (that sometimes coincide): the interface (static type of reference or pointer), and the implementation (the dynamic type of the object). This seems fairly straightforward to teach and reason about.

The following code example illustrates the proposed behaviour:

```
struct X {
  virtual void f() pre(a) post(b);
};

struct Y : X {
  void f() override pre(c) post(d);
};

int main() {
  Y y;
  y.f();  // checks c, then executes Y::f(), then checks d

  X& x = (X&)y;
  x.f();  // checks a, then c, then executes Y::f(), then checks d, then b
}
```

Note that under the Contract MVP rules for elision and duplication of contract checks ([P2900R6] sections 3.5.5. and 3.5.6), if the virtual call happens through the overridden interface but the compiler can prove that one contract subsumes the other (for example, the overriding preconditions are wider and/or its postconditions are narrower than the overridden ones), the compiler is allowed to elide the subsumed contract checks as a matter of QoI. At the same time, no such subsumption proofs are *required* to be performed.

5.3 Deep Inheritance Hierarchies

Intermediate base classes that are neither the statically called type nor the dynamic type of the object are not relevant for the correctness of the given virtual function call, so their precondition and postcondition assertions are not checked:

```
struct X {
  virtual void f() pre(a) post(b);
};

struct Y : X {
  void f() override pre(c) post(d);
};

struct Z : Y {
  void f() override pre(e) post(f);
};

int main() {
  Z z;

  X& x = (X&)z;
  x.f();  // checks a, then e, then executes Z::f(), then checks f, then b

  Y& y = (Y&)z;
  y.f();  // checks c, then e, then executes Z::f(), then checks f, then d
}
```

5.4 Multiple Inheritance

Precondition and postcondition assertions on a function overriding multiple functions are allowed; each function involved can have its own independent set of precondition and postcondition assertions. Just like before, base classes that are neither the statically called type nor the dynamic type of the object are not relevant for the correctness of the given virtual function call, so their precondition and postcondition assertions are not checked:

```
struct X1 {
  virtual void f() pre(a1) post (b1);
};

struct X2 {
  virtual void f() pre(a2) post (b2);
};
```

```
struct Y : X1, X2 {
  void f() override pre(c) post (d);
};

int main() {
  Y y;

  X2& x2 = (X2&)y;
  x2.f();  // checks a2, then c, then executes Y::f(), then checks d, then b2
}
```

5.5 Virtual Inheritance and Pure Virtual Functions

Precondition and postcondition specifiers on overridden functions in virtual base classes and on overridden pure virtual functions are allowed and behave in the same way as described above; no special rules are needed for these cases.

5.6 Member Function Pointers

As with regular function pointers, the same function-contract assertions that would be evaluated when invoking a function directly should be evaluated when invoking that function through a member function pointer. Also as with regular function pointers, there is no support currently proposed for such pointers to have function-contract assertions attached to their pointed-to function types.

```
struct S {
  virtual void f() pre(false);  // #1
};

void test()
{
  void (S::*ptr)() = &S::f;
  S s;
  (s.*ptr)();  // contract violation of #1
}
```

6 Implementation Strategies

6.1 Inline Interface Checks into Caller and Implementation Checks into Callee

Consider again the scenario from section 5.4, to establish nomenclature:

```
struct X {
  virtual void f() pre(a) post(b);
};

struct Y : X {
  void f() override pre(c) post(d);
};
```

```
int main() {
  Y y;
  y.f();
  X& x = (X&)y;
  x.f();
}
```

In the call `y.f()`, the "interface" (static type of `y`) matches the "implementation" (dynamic type of `y`), and in the call `x.f()`, the "interface" (`X&`) does not match the "implementation" type (`Y`).

Inlining interface checks into the caller is always possible, since they are determined based on the static type of the reference or pointer they are invoked on.

The line `x.f();` then, without optimization or QoI, executes as:

```
main:
  pre(a)
  Call x.f() through vtable
Y::f:
  pre(c)
  Body of Y::f()
  post(d)
main:
  post(b)
```

6.2 Check in Thunks

Strategy 6.1 could be implemented in statically-dispatched thunks instead:

```
main:
  Call x_f_thunk(y)
x_f_thunk:
  pre(a)
  Call y.f() via vtable
Y::f:
  pre(c)
  Body of Y::f()
  post(d)
x_f_thunk:
  post(b)
main:
  ...
```

This requires generating a thunk for each combination of overridden-overrider, but allows the compiler to select whether to inline the interface checks or not. Note that we already generate such thunks whenever we need to adjust the this pointer argument or the covariant return value of an overridden function, so this approach is not novel.

These thunks could be generated by the caller and weakly linked, or could be generated at the same time the class definition (and virtual function table) is generated, reducing the need to redundantly generate the thunks in many translation units.

Note that these thunks would possibly need to be more like full-featured function calls than thunks might be on some ABIs. Future features that might be adopted for Contracts, such as precondition captures or procedural interfaces (see [P2755R0]), will all require not just evaluating code before and after invoking the final overrider but also the allocation of stack space for the captures or local variables of the interface. Any ABI-changing implementation stragey must also take into account these future needs.

6.3 New Vtable Entries

When a virtual function has covariant return types, a new vtable entry (at least in the Itanium ABI) is added whenever an overriding function has a return type that needs an offset adjustment relative to the return type of the function it overrides. For virtual functions a similar approach could be taken by extending the vtable for each derived class that overrides a function. This vtable entry for each class would then be pointed at a thunk which checks the interface being invoked, followed by evaluating the target function and its function-contract assertions.

Adding function-contract assertions to a function would then be an ABI break, while changing them could be done without any need for clients to recompile.

Extending this option, new vtable entries could be added for all intervening classes, which would result in being able to always distinguish a different vtable entry for the full cross-product of interface types and virtual functions. In this case, no changes of function-contract assertions would be ABI breaks, and would just involve adjustments to the vtable contents and generation of thunks that checked the appropriate function-contract assertions.

6.4 Selective Implementation

The final strategy to consider is to document that the called interface function-contract assertions will always be evaluated with the *ignore* contract semantic. This option removes any burden on the caller, removes any need for recompilation when functions change, but significantly reduces the number of defects that will be detected by a platform that makes this decision.

6.5 ABI-stability Implications

In strategy 6.1, changing the contract on the interface requires recompiling all the callees calling through it to pick up the new contract; not recompiling the client code does not, however, render the program inoperable, it will just lead to a program where the new interface assertions are not checked. In strategy 6.2, contracts could be changed with a relink although adding contracts to a function that did not previously have them might again require recompilation. If history is any guide, making thunk symbols “extern” might be a feasible conforming extension, so that the thunk addresses could even be loaded from shared objects. Critically, the set of such symbols is enumerable; it is exactly one per virtual function declaration with an assertion specifier (or overrides one that does without any of its own).

Strategy 6.2 could be extended even further to *always* invoke such thunks, even when empty, allowing for relinking even when adding function-contract assertions to functions that did not previously have them. This would, however, have one of the largest overheads that might be considered.

Strategy 6.3 when first implemented would be a hard ABI break. In its most extreme option, when all potential combinations of invoking type and virtual function are placed into the vtable, further changes to the function-contract assertions of a function would no longer have any impact on the ABI of that function.

Strategy 6.4 is trivially not an ABI break and does not require recompilation, with the tradeoff that interface assertions are not checked. It bears strong similarity to strategy 6.1 when a callee is recompiled but a caller is not after function-contract assertions are added to the interface. The program will still likely work correctly, the function will be invoked, but the caller-side contract assertions will be ignored in that build.

7 Other Use Cases and Possible Future Extensions

7.1 Inheriting the Overridden Contract

Sometimes, the user might wish to actually inherit the preconditions and postcondition assertions of an overridden function (i.e. have them always checked when the overriding function is called, even if this call does not happen through the overridden interface). This straightforward approach of not altering the contract across a class hierarchy is likely going to be the first step in any implementation before a derived class contract is further refined, if it ever is.

Consider a class hierarchy with different algorithms that choose one number out of a non-empty set of numbers:

```
struct NumberPicker {
  virtual int pick(const Array<int>& numbers)
    pre (!numbers.empty())
    post (r: numbers.contains(r)) = 0;
}

struct FirstNumberPicker : NumberPicker {
  int pick(const Array<int>& numbers) override {
    return numbers[0];
  }
}

struct RandomNumberPicker : NumberPicker {
  int pick(const Array<int>& numbers) override {
    return numbers[rand() % numbers.size()];
  }
}
```

One normal use of these types would be to do so through the `NumberPicker` abstract interface, in which case the function-contract assertions of `NumberPicker::pick` will be evaluated:

```
int pickFromNumbers(NumberPicker& picker) {
    return picker.pick(getNumbers());
}
```

On the other hand, any code which knows the concrete type of the picker will avoid virtual dispatch through the base class and thus not evaluate any function-contract assertions, leaving the contracts of the derived class `pick` overrides unchecked:

```
int pickRandom(const Array<int>& numbers) {
  RandomNumberPicker picker;
  return picker.pick(numbers);  // plain-language contract not checked
}
```

This situation occurs more subtly when using templates instead of dynamic dispatch:

```
template <typename Picker>
int pickFromNumbers2(Picker& picker) {
  return picker.pick(getNumbers());
}

void testPicking() {
   FirstNumberPicker fnp;
   int i = pickFromNumbers2(fnp);  // plain-language contract not checked

   NumberPicker& np = static_cast<NumberPicker&>(fnp);
   int j = pickFromNumbers2(np);   // plain-language contract checked!
}
```

In order to address this use case with our proposal, the function-contract assertions of the overrides need to be repeated on the declarations of the overrides:

```
struct FirstNumberPicker : NumberPicker {
  int pick(const Array<int>& numbers) override
    pre (!numbers.empty)
    post (r: numbers.contains(r));
}
```

It might appear that this leads to checking the same condition multiple times when invoking through the base class — but that cost is minimal and should in general be elidable under the Contract MVP rules for elision and duplication of contract checks ([P2900R6] sections 3.5.5. and 3.5.6).

In cases where repeating the sequence of precondition and postcondition assertions on the overriding function presents an undue burden, it is possible to factor out these assertions into a separate function for re-use, or to refactor the code into a non-virtual function that asserts the preconditions and postconditions and then calls a virtual function containing the actual algorithm (also known as the "Non-Virtual Interface" idiom or NVI):

```
struct NumberPicker {
  int pick(const Array<int>& numbers)
    pre (!numbers.empty())
    post (r: numbers.contains(r)) {
      return pickImpl(numbers);
    }

private:
  virtual int pickImpl(const Array<int>& numbers) = 0;
}
```

```
struct FirstNumberPicker : NumberPicker {
private:
  int pickImpl(const Array<int>& numbers) override {
    return numbers[0];
  }
}
```

Should this option prove too burdensome, future evolution of the Contract facility could provide mechanisms to explicitly inherit the contracts of overridden functions. Such syntax must consider multiple inheritance and possibly whether function-contract assertions can be partially inherited, such as inheriting just preconditions or just postconditions from the overridden function.

7.2 Enforcing Contract Inheritance In The Base Class

Sometimes, one wishes to *enforce* in the base class that the precondition and postcondition assertions are inherited in all derived classes. Often, the motivation for such enforcement is to maintain some kind of class invariant to guarantee safety and/or correctness.

Consider a `Vehicle` base class and suppose we would like to maintain a speed that is within some speed limit:

```
struct Vehicle {
  virtual void drive(float speed_mph)
    pre (speed_within_limit())
    post (speed_within_limit());
}
```

Now, one might wish to enforce in this base class that no derived class can exceed the speed limit in their override of `drive`. Our proposal does not offer such enforcement. This is on purpose: enforcing it would preclude a number of important use cases (see Section 3). We could certainly entertain a syntactic marker to opt into such enforcement as a future extension; nothing in this proposal precludes this direction. Meanwhile, it is possible to use NVI (see example in the previous section) or, better, to encapsulate the invariant in the base class such that it cannot be modified by the derived class:

```
struct Vehicle {
  virtual void drive(float speed_mph);

protected:
  void setSpeed(float speed)
    pre (speed_within_limit())
    post (speed_within_limit());

private:
  // cannot be accessed by derived classes
  float current_speed = 0;
  bool speed_within_limit() const { return current_speed <= limit }
}
```

The precondition and postcondition assertions in [P2900R6] are primarily designed to validate the preconditions and postconditions specific to the function they apply to. They could be used to validate class invariants, but that is not their goal.

Validating class invariants should be addressed by a future extension explicitly designed for this purpose. The most plausible design for such an extension is a new kind of contract assertion placed in the class definition (discussion of such a design can be found in [P2755R0] Section 2.2.11 and [P2885R3] Section 7.11). While such invariant assertions can be naively thought of as "postconditions of all constructors, preconditions of all destructors, and pre/post conditions of all other member functions", there are major questions that need answering related to them to have a complete design: Should these be checked on `private` member functions? Should these be checked on `const` member functions (and if not, how to deal with `mutable` members)? Should these be checked on member functions called from other member functions? How to avoid recursive checks in all these cases? etc. Not all of these questions may have universal answers.

In any case, we do not think invariants come into play when considering what should be checked on a virtual function invocation. They come into play in terms of how we map a type's declared invariants to what we consider to be the function-contract assertions of a given function — but that then feeds into the virtual dispatch rules that this proposal is designed to be consistent with, and does not need to be treated specially by those rules.

7.3 Checking Base Class Contract Only When Running Base Class Implementation

Occasionally, a virtual function intended to be overridden also provides a concrete implementation — perhaps as a default, or perhaps as a utility to aid in implementation of overrides. In this case, the base class implementation might provide a set of preconditions and postconditions that should only be asserted when that base class implementation is called, never when a derived class implementation is called (even if the latter is called through a pointer or reference to base). Consider:

```
struct Shape {
  virtual void draw()
  // this base class implementation only works when testMode == true
  { /* ... */ }
}

struct Circle : public Shape {
  void draw() override
  // no preconditions – this implementation works always
  { /* ... */ }
}

void drawShape(Shape& shape) {
  shape.draw();
}

void drawACircle() {
  // testMode == false
  Circle myCircle;
  drawShape(myCircle);   // this should work!
}
```

In this example, the check for `testMode == true` is conceptually not part of the interface of `Shape::draw`, but only a part of its implementation, so it cannot be expressed with a `pre` on `Shape::draw`. However, it can instead be easily expressed with a `contract_assert` statement inside the body of `Shape::draw`:

```
struct Shape {
  virtual void draw() {
     contract_assert(testMode);
  }
}
```

Such a set of preconditions and postconditions that should only apply to a particular implementation might be provided in addition to the preconditions and postconditions that are expected of all instances of the class when called through a pointer or reference to base.

For example, an abstract class for random number generation might provide a trivial implementation to ease initial implementation:

```
struct RandomGenerator {
  virtual int next()    // generate a random number in the range [0,100).
  { return 0; }
};
```

This generator function might want to provide two sets of preconditions and postconditions. When invoked virtually, the postcondition that the result is in the range $[0, 100)$ should be universally applied. When this particular implementation is invoked, a stronger postcondition that the result is 0 can be asserted.

In this case, we can express the set that is conceptually part of the interface with `pre` and `post`, and the set that is conceptually part of the implementation with `contract_assert`:

```
struct RandomGenerator {
  virtual int next()
    post( r : r >= 0 && r < 100 )
  {
    auto result = 0;
    contract_assert( result == 0 );
    return result;
  }
};
```

The disadvantage of `contract_assert` is that it is not syntactically part of the interface, which might be desirable in this case to let the users of the function know about the contract on its implementation. As a future extension, we could entertain a new shorthand syntax such as a label that attaches to function-contract assertions and makes them evaluate only when the particular function implementation is invoked (essentially replicating Ada's dual model of *specific* and *class-wide* contract assertions). Such assertions specific to an implementation could not be depended on by clients of the function using virtual dispatch, but would increase expressivity for situations like this.

7.4 Static Analysis

Static analysis of the correctness of contract assertions is an important use case for the feature to support. In general, our approach has aimed to maximize the amount of local reasoning a static analyzer can do.

When invoking a function through dynamic dispatch, the function-contract assertions on the (base-class) function being invoked are visible, are expected to be upheld, and no knowledge of the dynamic type is necessary. A static analyzer can verify that the preconditions of that function are being satisfied, and can leverage the postconditions for further reasoning. Regardless of the dynamic type, when used through a particular base the requirements and guarantees of that base class function will be expected for the call.

On the other side, a static analyzer can address the correctness of a function by looking at only that function's precondition and postcondition assertions and identifying if they are consistent with the function's implementation.

Finally, a static analyzer will be able to look at each override and determine if its interface is consistent with the base class function — possibly producing warnings if a function has narrower preconditions than one it overrides, or if it might fail to satisfy an overridden function's postconditions when its preconditions are satisfied. As with almost all static analysis that is no whole-program analysis, such warnings would be exactly that — warnings — as a defect only occurs when objects of that type are used in situations that don't externally guarantee that the required additional conditions are satisfied. This might seem sub-optimal, but the ability to abstract such concerns to larger scopes and separate them from local algorithms is one of the major strengths of object oriented programming, and though that comes at the cost of local provability, it enables useful software engineering.

8 Proposed Wording

The proposed wording is relative to [P2900R6].

Modify [dcl.contract.func]:

> A coroutine ([dcl.fct.def.coroutine]), ~~a virtual function ([class.virtual])~~, a deleted function ([dcl.fct.def.delete]), or a function defaulted on its first declaration ([dcl.fct.def.default]) may not have a *function-contract-specifier-seq*.
>
> [...] [*Note:* The precondition assertions of a function are evaluated in sequence when the function is invoked ([intro.execution]); in a virtual function call ([expr.call]), the precondition assertions of the statically chosen function are evaluated first, followed by those of the final overrider. The postcondition assertions of a function are evaluated in sequence when a function returns normally ([stmt.return]); in a virtual function call, the postcondition assertions of the final overrider are evaluated first, followed by those of the statically chosen function. *— end note*]

Modify [expr.call], paragraph 6:

> When a function is called, each parameter ([dcl.fct]) is initialized ([dcl.init], [class.copy.ctor]) with its corresponding argument and each precondition assertion

> ([dcl.contract.func)] is evaluated; in a virtual function call, the precondition assertions of the statically chosen function and then those of the final overrider are evaluated. 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:

> The postfix-expression is sequenced before each expression in the expression-list and any default argument. The initialization of a parameter, 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, which are evaluated in sequence ([dcl.contract.func]). In a virtual function call, the precondition assertions of the statically chosen function are evaluated in sequence; these evaluations are sequenced before evaluation of the precondition assertions of the final overrider, which are evaluated in sequence.

Modify [stmt.return]:

> All postcondition assertions ([dcl.contract.func]) of the function are evaluated in sequence. In a virtual function call ([expr.call]), the postcondition assertions of the final overrider are evaluated in sequence; these evaluations are sequenced before evaluation of the postcondition assertions of the statically chosen function, which are evaluated in sequence. The destruction of all local variables within the function body is sequenced before the evaluation of any postcondition assertions. [*Note:* This in turn is sequenced before the destruction of function parameters. — *end note*]

Wording note: the above changes are a minimal set of changes upon the current state of [P2900R6]. It is possible that a bigger restructuring, which moves all mention of postcondition evaluation to [expr.call] and thus to a place where there is full awareness of the virtual nature of the particular function call, would be a better solution, but that will be resolved when this wording is merged into the rest of the Contracts wording.

9 Conclusion

C++ with Contracts will be a powerful tool for writing correct code. We believe it will be immensely useful for the community even without support for virtual functions, but proper support for contract checking on virtual functions will further increase the viability and robustness of the Contracts feature in C++.

To that end, we have proposed a simple rule — that we believe is both minimal and sufficient — for which function-contract assertions are evaluated when invoking a function through virtual dispatch. The function-contract assertions of both the base class virtual function being used by the caller and those of the overriding function that is actually invoked will be evaluated. Beyond that, no virtual function has implicit function-contract assertions, or restrictions on what function-contract assertions it may have.

While any design in this space is ultimately a tradeoff between making certain use cases more convenient at the expense of others, we believe that our proposal enables the widest set of use cases overall while maximizing compatibility with the most common and useful usage patterns and consistency with the underlying mechanics of virtual functions in C++. Most importantly, both callers and callees will have their expectations checked when invoking a function, while flexibility exists to vary those expectations or work around them wherever needed.

Acknowledgements

Thanks to the Ville Voutilainen, Lisa Lippincott, Herb Sutter, Ran Regev, Tom Honermann, and Andrzej Krzemieński for discussing and reviewing parts of this paper.

Bibliography

[N1613] Thorsten Ottosen, "Proposal to add Design by Contract to C++", 2004
http://wg21.link/N1613

[N1669] Thorsten Ottosen, "Proposal to add Contract Programming to C++ (revision 1)", 2004
http://wg21.link/N1669

[N4110] J. Daniel Garcia, "Exploring the design space of contract specifications for C++", 2014
http://wg21.link/N4110

[N4415] Gabriel Dos Reis, J. Daniel Garcia, Francesco Logozzo, Manuel Fahndrich, and Shuvendu Lahri, "Simple Contracts for C++", 2015
http://wg21.link/N4415

[P0247R0] Nathan Myers, "Criteria for Contract Support", 2016
http://wg21.link/P0247R0

[P0380R0] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup, "A Contract Design", 2016
http://wg21.link/P0380R0

[P0542R5] J. Daniel Garcia, "Support for contract based programming in C++", 2018
http://wg21.link/P0542R5

[P2388R4] Andrzej Krzemieński and Gašper Ažman, "Minimum Contract Support: either No_-eval or Eval_and_abort", 2021
http://wg21.link/P2388R4

[P2755R0] Joshua Berne, Jake Fevold, and John Lakos, "A Bold Plan for a Complete Contracts Facility", 2023
http://wg21.link/P2755R0

[P2885R3] Timur Doumler, Joshua Berne, Gašper Ažman, Andrzej Krzemieński, Ville Voutilainen, and Tom Honermann, “Requirements for a Contracts syntax”, 2023
http://wg21.link/P2885R3

[P2900R5] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R5

[P2900R6] Joshua Berne, Timur Doumler, and Andrzej Krzemieński, “Contracts for C++”, 2024
http://wg21.link/P2900R6

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