Contracts do not belong in the language

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P3929R0

Working Group: Library, Core, Evolution

Date: 2025-09-02

David Chisnall <David.Chisnall@cl.cam.ac.uk>

John Spicer <jhs@edg.com>

Ville Voutilainen <ville.voutilainen@gmail.com>

Gabriel Dos Reis <gdr@microsoft.com>

Jose Daniel Garcia Sanchez <josedaniel.garcia@uc3m.es>

Introduction

Contracts have been proposed in P2900 as a new language feature. This paper argues that, assuming that contracts are a desirable feature, implementing them in the language as described by P2900, is the wrong approach.

When should we add language features

Before arguing that a feature does not belong in the language, it is worth taking some time to explore why features *should* be added to the language. It is unlikely that any member of WG21 believes that there are no possible future improvement for C++ and that the language should be frozen. Some common vocabulary for evaluating when features belong in the language is useful.

New abstract-machine behaviour or new concrete representations require language features

If a feature cannot be expressed at all, either in the C++ abstract machine or in a way that permits lowering to a desired output on concrete targets, the only possible solution is to extend the language. This can be either an extension to the standard or a vendor-specific extension.

The introduction of `thread_local` in C++11 fell into this category in both respects. Without `thread_local`, it was impossible to provide things with a globally visible name, but with a per-thread copy, within the abstract machine. Most C++ targets in 2011 provided a notion of thread-local storage and there was no way of writing C++ source code (without vendor extensions, such as GNU C++'s `__thread`) that exposed this underlying concrete-machine feature.

When a problem is in this space, there is no question of whether it is a language or standard-library feature: it cannot be implemented in the library. The only question is whether it is a desirable feature.

When adding a feature of this kind, it is essential to clearly specify the semantics. If a feature is wrapping some existing portable behaviour provided by underlying platforms then this can be delegated to the ABI to describe how the mapping is done. If the semantics are new, they must be clearly expressed as extensions to the abstract machine.

Difficulty to expressing common idioms succinctly motivates language features

Some language features are *syntactic sugar*. This is not a dismissive term, syntactic sugar is what makes a language easy to swallow.

Lambdas are the clearest example of recent syntactic sugar added to C++. C++11 lambdas provided nothing that that you could not express by writing a class that had fields for each capture, a constructor that initialised them, a call method, and so on. Later extensions followed the same pattern, defining templated call operators and so on. The behaviour of any lambda expression can be expressed in terms of a class. The lambda form is *far* more concise and so enables patterns that would have been too verbose for people to write without lambdas, even though they were technically possible.

New language features of this kind are easier to specify because they can be described as source-to-source transformations (even if they are never implemented in this way). Some, such a range-based for loops, are very simple transforms, yet have disproportionately large (positive) impacts on readability.

New language features make C++ harder to learn

'C++ is too complicated' is a common argument from people who do not use the language. Most new language features strengthen this argument. Even the most useful language feature will increase complexity for new developers.

Removing features from a language with as many users as C++ is almost impossible. Even removed standard-library features are typically still supported by implementations.

Removing language features is harder. C++17, for example, removed trigraphs, a feature that has been largely unnecessary since before ISO C++ was standardised. Very few other language features have ever been removed from C++. Once something is added, it should be assumed to be there forever.

New language features constrain future language features

Language features do not exist in isolation. Each language feature interacts with other parts of the language. When a new feature is introduced, it is possible to reason about how it interacts with existing features.

Every new feature makes this reasoning harder because the set of things that the next feature may interact with is larger. This imposes a burden on future language evolution.

Again, this is not to say that we should *never* extend the core language, only that new features should be carefully considered. Current in-flight proposals such as `match` expressions and string interpolation, for example, have been demonstrated in other languages to significantly simplify complex codebases. These are clear benefits to the language from such features (making no comments on the specific C++ integrations of these features in the papers that propose them).

The complexity budget for new features should be considered a scarce resource and spent carefully.

What kind of language feature is contracts?

Contracts, as proposed in P2900, are introducing both new abstract-machine semantics *and* syntactic sugar. It includes:

- New semantics for things that are currently violations of the one-definition rule (ODR)
- `pre` and `post` as ways of writing code that runs before and after functions
- `contract_assert` as a function-call-like statement within a function
- A (replaceable) global hook that is invoked on failure

The new semantics are complex (more on this later).

The new syntax is almost possible to define in terms of existing abstract-machine semantics but this is not the approach taken in P2900. This is motivated by a desire to allow different implementations but the result is that almost everything in P2900 is implementation-defined behaviour. This makes it very hard to reason about the semantics of a program at a source level.

There has been some discussion of whether contracts do or do not change ODR. This arises from the fact that ODR is not addressed directly in P2900, only via implication. It is (as outlined in the 'Mixed Mode' section) permitted for two compilation units to be built with two different contract enforcement modes. This means that pre- and post-conditions and, most importantly, `contract_assert` can differ in two versions of a function from a header shared between the two.

This composes with the rule that it must be possible to adopt contracts in a compilation unit in a program where not every compilation unit is compiled with a contracts-aware compiler. This means that, if `foo()` is a function containing a `contract_assert` then it must have the same decorated name in all compilation units, irrespective of their contract mode, otherwise callers would break. It may be possible to define `inline` functions that have contracts to use different decorated names, because it would already be an ODR violation if the `contract_assert` were `#define`d away, though this was not done in the implementations from the implementation report.

This makes contracts unique in how the affect lowering in a modern C++ compiler. The front end parses the source into an abstract syntax tree (AST) and then lowers it to some intermediate representation for mid-level optimisers. This lowering depends on the token stream (including pre-defined macros, which are a short-hand way of adding tokens to the front of a compilation unit) and the target and, prior to P2900, nothing else (for standard C++: vendor extensions such as `-ffast-math` may affect this).

Consider the following example:

```
inline foo(X *arg)
{
    if constexpr (SOME_MACRO)
    {
        if (arg == nullptr)
        {
            abort();
        }
    }
    ...
}
```

Compiling this with two different values of `SOME_MACRO` would violate ODR. If the `if constexpr` were replaced with `#if SOME_MACRO` then the same would apply. This is an important property of ODR because this violation would change the *pre-optimisation* semantics of the function and so it is impossible for any later optimisation to reason about the behaviour of the two different variations of this function when it has access to only one.

Now consider this version after P2900:

```
inline foo(X *arg)
{
    contract_assert(arg != nullptr);
    ...
}
```

It is permitted to mix versions of this with different contract semantics and the linker is required to pick one. There is no other way of writing `foo` without contracts that would lower to something equivalent conditional on a compiler setting and not be an ODR violation. This is one of the key arguments for contracts being part of the language made in P2900.

Whether intentional or not, this is a relaxation of ODR in a way that fundamentally alters the guarantees that a C++ front end provides to (mostly language-agnostic) mid-level optimisers.

Are contracts a good use for the complexity budget?

Disclaimer: This section contains some deeply unscientific informal polls. The results should be considered vaguely indicative, do not take the absolute numbers as statistically significant.

As a finger-in-the-air feel, I posed the following poll:

C++ contracts, are they important to you?

- I use C++ (21%)
- I don't use C++ because it doesn't have contracts (2%)
- I don't use C++ and adding contracts will not change my mind (77%)

The results are with 86 respondents (the poll is still open), almost four fifths are people who do not use C++. The number of people who would adopt C++ if it had contracts is below the lizardman constant (the number of people who give nonsense answers to surveys, usually estimated at 4%, sometimes higher).

This suggests that, although contracts may be very important to some users, they are not a widely desirable feature.

Given the number of responses from people who are not C++ programmers, I tried another poll to determine why people don't use C++ (results with 532 respondents):

- It is not memory safe (22%)
- It is too complex (73%)
- It lacks a feature I consider important (2%)
- It doesn't work on the target I care about (3%)

This suggests that, although adding contracts may make around 2.5% of people consider adopting C++ (the respondents from the first poll who were not C++ programmers because it lacks contracts), 73% are already put off by the complexity, something that contracts makes worse.

The answers about missing features followed up in the comments and were related to dependency management and business-language features such as GUI or web-app frameworks with low boilerplate. Some of these, particularly those related to web apps, *could* be addressed with generic function decorators (more on this later).

Problems with P2900

As a concrete proposal, there are several issues that I argue make P2900 inappropriate as a new language feature for C++26.

P2900 is underspecified

By design, P2900 leaves much to be implementation defined. This includes whether contract checks are inlined in the caller or callee, how contract enforcement modes are selected, the interactions between different compilation units builtin different modes, and so on.

This leads to some difficulties in reasoning about semantics. For example, consider the following function in a header file:

```
inline int* f(int *x)
{
  contract_assert(x != nullptr);
  return x;
}
```

A caller writes:

```
int foo(int *x)
{
    if (f(x) == nullptr)
    {
        return 0;
    }
    return *x;
}
```

If this function is compiled with quick-enforce semantics and then linked to another compilation unit that is compiled with the ignored semantics, what happens? In the current GCC version, the linker may select a version of `f` that omits the contract check. At the same time, the caller assumes that the contract check happens and gcc elides the check.

This is similar to the issue that caused security vulnerabilities in the Linux kernel, where null checks were elided because the compiler could prove that the pointer was used on a path not dominated by the check. This particular example would be a powerful tool for supply-chain attacks. An attacker simply needs to add a contract assertion into a header that accompanies a shared library, built with contracts ignored, and then convince the authors of a security-critical piece of software that they should build with quick-enforce semantics for extra security. All parts of the program are correct and secure in isolation, only the composition sees the security vulnerability.

In the general case, this optimisation is desirable. Consider the following alternative definition of `f`:

```
inline int* g(int *x)
{
  static int y;
  if (x == nullptr)
  {
    x = &y;
  }
  return x;
}
```

In this case, eliding the null check from a caller would be sound. It would provide a reduction in code size and an increase in performance.

It is difficult to design a compiler mid-level representation that permits the latter transform but not the former. Doing so requires tracking, within a function, what things may and may not be assumed in the presence of replacement, at a granularity that no other language feature (in C++ or other languages supported by GCC or LLVM's optimisers and back ends) currently requires.

If the syntactic additions from contracts fitted cleanly into the second category of addition (syntactic sugar) it would be easy to decompose the parts that are syntactic additions and those that require new semantics. The current paper is intended to permit future linker behaviours but these have not been prototyped and may require ABI-breaking changes.

P2900 introduced new single-use syntax and semantics

The new syntax in P2900 (`pre`, `post`, and `contract_assert`) are all special cases of generically useful behaviour. The pre- and postcondition hooks allow running code before and after functions.

This is a very useful feature, beyond contracts. Python, for example, provides a generic feature for this: decorators. These are functions that take a function as an argument and modify it. It is used for a wide range of things, including memoisation, automatic logging, access control, serialisation, and so on.

Assuming reflection is added to C++26, there will be building blocks for creating this (compile-time decorators) as a generic language facility. It's not clear that this will be ergonomic with reflection, but the proposal to allow reflection over attributes should provide a building block. A later addition could then provide syntactic sugar for concise contracts.

Creating contracts in this way would have one key advantage: we would begin standardising concise syntax with a corpus of examples of how people wish to use contracts. The examples that exist today and that informed P2900 are all drawn from contracts systems with different semantics. By providing building blocks with the same semantics as the final version, we could optimise the syntax based on examples that could be accurately expressed before the final syntax is standardised.

Similarly, the ODR relaxations are broadly useful. There are several examples where libraries use non-standard always-inline attributes to avoid ODR violations that could be replaced by a standard behaviour that says that a function may be replaced by other non-identical versions at link time, but that it's acceptable to link the current version into the binary as long as the generated version is not exposed as a symbol. Contracts, ideally, want a relaxation of this general facility that constrains the space of possible changes (as do other things, such as versions with verbose logging). Building this separately would allow it to be prototyped independently and the consequences fully understood before being placed in the standard.

Note that contracts in P2900 are *not* simply decorators because the syntax is tightly coupled to the non-ODR-violation semantics. This makes it hard to explain the semantics to programmers and hard to reason about them at the source level.

Eliminating the non-ODR-violation semantics would require that, in the MVP, all parts of a program must be compiled with the same contract semantics (the same is true for macro- or `constexpr`-based invariants frameworks today, so this would not be a regression, simply a lack of improvement). With that omitted; however, P2900 is a specification for special-case decorators.

P2900 lacks important features for contracts

P2900 is described as a minimum viable product. As such, it lacks key features that existing invariants frameworks provide. These include, but are not limited to:

- Per-subsystem semantics, so that you can control checks on individual parts of a program. The clang prototype includes this as an extension.
- Per-severity reporting, so that individual deployments can choose to trade performance overhead of dynamic checks with safety at more than a binary granularity.
- The ability to handle violations differently (for example, a library shipped as a binary may wish to provide its own telemetry hook for internal violations, but use the caller's hook for API misuse errors).
- Concise syntax for reusable checks, such as non-null parameters.
- Providing useful debugging messages on contract failures.

Some of these can be implemented in backwards-compatible ways at the source level, some will require modification, some may break ABIs. Other papers are proposing to treat other kind of failure as contract violation, which would make decomposing the single global handler for contracts difficult. We have learned from `std::unexpected` that single global handlers are difficult to use in practice for erroneous conditions.

Some other features are omitted and are not present in most existing invariant frameworks but would be useful, such as, the ability to apply contract checks to all classes that implement a concept. A concept provides a shape for a set of classes as a structural type. Being able to specify contracts in concept definitions and then either synthesise a wrapper or explicitly opt into the concept to apply them to the class would make concepts more useful. It is not clear whether P2900 can be extended in such a way.

Composition with future desirable features

Deep immutability is a generically useful concept in a language. C++ does not currently implement such a concept because it depends on viewpoint adaptation in the type system (roughly: the type of a field of an object depends on the type of the pointer to that object that you followed to get to it). In the evolution of P2900, it became clear that this would be desirable for contracts. The arguments to preconditions are made `const`, but this provides shallow immutability: a `const` member function may mutate something reachable from the object.

Contracts are not the first, or only, place where deep immutability would be desirable. If this is added in C++29 or C++32, would contracts be modified so preconditions take a `deep_const` view of arguments? Would we need a new version of `pre` with a new core-language keyword to describe it?

As previously mentioned, generic decorators are a phenomenally useful feature. The existence of `pre` and `post` makes these harder to specify. Do decorators see the version of the function including pre- and post-conditions? Do they see the unwrapped version? Do preconditions apply before a decorated function?

By introducing special-case syntax and semantics in a place where general-purpose syntax and semantics are desirable, P2900 makes it harder to implement the general case.

Decomposing the requirements of contracts

The remainder of this paper proposes straw-man sketch of a decomposed set of C++ language extensions that could be combined to build a richer set of contract APIs than P2900.

Safe ODR exclusions

Currently, compilers use ODR to enable optimisations. Whether this is sound is a much longer discussion and the answer is complicated. For the purpose of this section, we will assume that there is at least a set of sound optimisations that are permitted by ODR.

The intent of ODR is that independent compilation units may emit copies of templated or inline functions that have no single canonical home. They may appear in multiple object files, static libraries, or even dynamic libraries. Each compilation unit can be optimised independently. The compiler may assume that the final program will contain either the version of the function that it can see and analyse or something semantically equivalent.

P2900 relaxes this to say that the mode switching of contracts does not violate ODR and so a compiler must be able to handle functions with contracts being replaced with ones that have the same text but different lowering of the contract checks. This is very useful for contracts. It permits linking debug libraries with release executables, and vice versa. It is also a useful feature in general for being able to clearly define coupling at library boundaries.

There is a reason that ODR has been part of the language for so long: relaxing it in a way that does not make optimisations unsound and still *permits* optimisation is a **very** hard problem. Unfortunately, P2900 simply declares that it is relaxed but that the behaviour of the relaxation is implementation defined. There are three possible implementation choices:

- Decorate names differently based on contract mode. This then makes pointer comparison complicated because the address of two different functions with different modes is different, yet pointers to them must compare equal. This is specific to the P2900 approach to contracts and is not intrinsic to the problem.
- Disable inter-procedural analysis of any functions that contain contracts. Compiler intermediate representations have existing mechanisms that permit this but the performance impacts of using them can be significant.
- Design some new compiler machinery to handle this behaviour, which is a very high cost for a single feature.

Given the benefits of being able to relax ODR and the extreme complexity of doing so soundly, there is clear value in decoupling this. It would initially allow easy evaluation of the cost of the big-hammer approach of blocking inter-procedural analysis. It would then allow exploration of the ways that this could be safely relaxed without significant performance impacts.

Function decorators

This paper has mentioned function decorators in several places. Decorators, broadly, take a function and wrap it.

For example, you could imagine defining a C++ decorator as:

```
template<auto Decorated, typename Return, typename... Args>
Return myDecorator(Args... args)
{
    // The types of the template arguments and the decorated function must match.
    static_assert(std::is_same_v<
        std::remove_reference_t<decltype(Decorated)>, Return(Args...)>);
    if constexpr (DynamicNullChecks)
    {
        std::tuple{args}.visit([](auto &x)
            {
                if constexpr (std::is_pointer_v<
                    std::remove_reference_t<decltype(x)>>)
                {
                    if (x == nullptr)
                    {
                        unexpected_null_pointer_argument(fn);
                    }
                }
            });
    }
    auto ret = fn(std::forward<Args>(args)...);
    if constexpr (DynamicNullChecks)
    {
        if constexpr (std::is_pointer_v<Return>)
        {
            if (ret == nullptr)
            {
                unexpected_null_pointer_result(fn);
            }
        }
    }
    return ret;
}
```

This (ignoring issues with accidental copies in the tuple construction, which are addressed by pack indexing and other in-flight features) is a generic decorator that could be applied to any function to assert that its pointer arguments are all non-null and that its result (if a pointer) is non-null. The checks happen only if the `constexpr` global `DynamicNullChecks` is true, allowing individual checks to be enabled and disabled at a fine granularity. Other generic checks could be built this way.

The same mechanism could be used to provide automatic logging and so on.

A generic C++ decorator feature would depend on the C++26 reflection. Ideally it would allow:

- Specifying decorators inline in headers for out-of-line function bodies.
- Writing generic decorator functions and applying them with a single attribute to any function or member function.
- Writing generic decorator functions and applying them with a single attribute to every member function in a class.
- Applying decorators to member function overrides in all subclasses.
- Defining decorators for concepts and generating wrappers that have them.
- Defining decorators for concepts and explicitly adopting them in a class that intends to implement that concept.
- Specifying whether the decorator should appear in the decorated name of the method, so that the decorator can be considered part of the implementation or a variation on the method.

A decorator feature that met these requirements would be a useful building block for a great many features (including higher-level things such as automatic bindings for RPC frameworks) and would enable a rich design space for contract implementations.

Given a decorators API, there is an incremental path to something equivalent to the current P2900 syntax:

1. A reflection-based decorators API that works.
2. Wrappers that take the wrapper functions that perform contract checks as lambdas and apply them to the decorated function.
3. Special-case attributes for contracts that provide concise syntax for contracts.

Each stage in this process would provide time for evaluating real-world use cases. In contrast, P2900 commits the standard to new syntax and semantics before any large-scale experiments can be run.

The initial version could be an attribute that applies a decorator such as the above example to the function. For example:

```
[[decorate(myDecorator)]]
void someFunction(SomeType *x);
```

This would be defined to replace `someFunction` with `myDecorator<someFunction, void, SomeType*>`. This is not desirable end-state syntax, but provides a convenient building block. This is a simple reflection-driven transform and does not require new bespoke syntax for a single use case, nor does it require new abstract-machine semantics.

The next extension would be to allow free-standing lambdas in attributes, permitting something like:

```
[[decorate([](void(auto decorated, SomeType *x) {
    if constexpr (MyCheckIsEnabled)
    {
        if (x == nullptr)
        {
            handle_invalid_null_argument();
        }
    }
    decorated(x);
    if constexpr (ReallyExpensiveChecksEnabled)
    {
        run_global_integrity_check();
    }
})]]
void someFunction(SomeType *x);
```

This would replace `someFunction` with the provided wrapper, but can be expressed as a source-to-source transform (no new abstract-machine semantics) in terms of the first decorator. Note that this is already more flexible than P2900 *for the specific case of contracts* in a variety of ways:

- It allows different handlers to be called for different classes of errors.
- It allows different checks in the same contract to be toggled independently.
- It allows common checks to be factored out, as in the check above that asserts that no pointer parameters are null.

Finally, if pre- and post conditions are generically useful (as shown by people using them in real-world code), the standard library can provide `pre` and `post` hooks built on this model. Importantly, other libraries can *also* provide different models for contracts that address their concerns directly without either conflicting with the default (and useful to most people) implementations, or requiring the default implementation to be useful to everyone.

Lazy execution

The C `assert` macro implicitly evaluates its arguments only if assertions are enabled. If `contract_assert` were implemented as a normal C++ function, the semantics would be different. Consider the following implementation:

```
void contract_assert(bool condition)
{
    if constexpr (!ContractsIgnored)
    {
        if (condition)
        {
            handle_violation();
        }
    }
};
```

If a user writes `contract_assert(someExpensiveToComputerCondition())` then the expensive computation would be performed and then passed to the assert function, which would then do nothing. This is one of the motivating reasons to place contract assertions in the language.

In language such as Haskell, with lazy evaluation, this is not a problem. The equivalent function receives a *thunk*, a wrapper that, when called evaluates the condition. This would be possible in C++ as a simple transform (which can be expressed as a source-to-source transform) enabled by an attribute. For example:

```
void contract_assert([[lazy]] bool condition)
{
    if constexpr (!ContractsIgnored)
    {
        if (condition)
        {
            handle_violation();
        }
    }
};
```

The example call from earlier is now implicitly transformed (in a similar way to lambda) into the equivalent of:

```
class
{
    bool cache;
    bool isFirstCall = true;
    public:
    bool operator()
    {
        if (isFirstCall)
        {
            isFirstCall = false;
            cache = someExpensiveToComputerCondition();
        }
        return cache;
    }
} __contract_assert_argument_thunk;
contract_assert(__contract_assert_argument_thunk);
```

This is similar to `[&]() -> bool { someExpensiveToComputerCondition(); }` but with memoisation applied such that the value is computed on the first call. Within the `contract_assert` function, each access of `condition` is replaced with an invocation of the thunk's call.

Deep immutability

Deep immutability is useful for contract checks because, ideally, preconditions would not accidentally mutate the arguments. It is more broadly useful. For example, in fork-join-style structured concurrency, it is useful to allow parallel iterations to have an immutable view of the same object graph but mutable views of independent objects.

Deep immutability would require marking member functions with a variant of `const` (such as `deep_const` that enforces stricter requirements:

- No fields of the object may be directly modified.
- All fields are treated as `deep_const` (only member functions that are marked as `deep_const` on them may be called).
- No global variables may be written.
- Global functions that take the object or its fields as arguments must take a `deep_const`-qualified pointer / reference.
- Global functions called must also be marked as `deep_const`, and may not mutate globals or call other functions that are not marked `deep_const`.

These are similar to the `consteval` requirements and there may be some overlap.

Conclusion

The contracts proposal in P2900 introduces a lot of bespoke complexity but all of the places where it introduces language-level is a special case of a feature that has generic value. Decomposing the desired semantics into independent features would allow them to be developed in parallel, shipped in experimental form in `-std=c++2d`, and evaluated against multiple use cases.

Each of these features would be useful for contracts and would enable a richer contracts API to live in the standard library. It would also allow third-party contracts APIs that address use cases that only a handful of projects have to coexist and reuse the same machinery, rather than requiring the language-provided contracts mechanism to address the long tail of requirements.

P2900 may be the best that we can do right now. The same was true of `std::auto_ptr` when it was introduced but the right fix was to add r-value references to the language and define move semantics. Getting from `std::auto_ptr` to `std::unique_ptr` was not a simple tweak. If `std::auto_ptr` had been added as a language feature then deprecating and removing it would have been far harder once a better approach became possible.