Outstanding design questions for the Contracts MVP

Timur Doumler (papers@timur.audio)

Document #: P2896R0
Date: 2023-08-22
Project: Programming Language C++
Audience: SG21

1 Motivation

As SG21 (the Contracts study group) moves along towards our plan [P2695R1] to build consensus on a minimal viable product (MVP) for a Contracts feature for C++, the major piece still missing from the design is the choice of syntax for contract-checking annotations (CCAs); see [P2885R0] and references therein. However, besides this choice of syntax, there are a number of other outstanding design questions that, while being less fundamental than the syntax question, still have to be answered by SG21 before we can put together a complete proposal for a Contracts MVP and forward it to EWG and LEWG for review.

This paper attempts to collect all these remaining outstanding design questions. We omit any questions that we believe can be considered as a post-MVP extension. We include only those that need to be answered before the Contracts MVP can be considered complete and correct, or represent serious known concerns about the current state of the MVP. All the questions in this paper will need to be answered by SG21 via polls, ideally accompanied by papers. According to the roadmap in [P2695R1], we should resolve all of these by the Spring 2024 meeting in Tokyo. This currently seems like an achievable goal.

Until now, open design questions for Contracts have been tracked in the "Controversial aspects" section of [P2521R4]. Recently several new design questions have appeared and have been actively discussed on the SG21 reflector. We therefore felt that it would be better to have a separate paper dedicated to keeping track of all currently known open design questions targeting the Contracts MVP: an MVP "to do list", if you will. This is this paper. We intend to keep the information presented here up to date by providing frequent revisions. Any given revision should be understood as a snapshot in time; we expect several questions listed here to be resolved soon, and others might surface that we are not yet aware of.

One design question listed here has been inherited from [P2521R4]; others have been discussed in more detail in other papers (to which we provide references below); yet others have only been discussed informally on the SG21 mailing list and are being published here for the first time.

2 Open design questions

2.1 Contracts on first declarations in different TUs

Status: Unresolved. Needs paper.

The current Contracts MVP specifies the following (see [P2521R4] section 3.11):

> "If a given function `f` has declared preconditions and postconditions, they shall be visible in the first declaration of `f` in a translation unit (TU), otherwise the program is ill-formed. Subsequent declarations must omit contract annotations. If `f` is declared in more than one TU, the corresponding first declarations of `f` shall be identical (modulo parameter names), otherwise the program is ill-formed with no diagnostic required."

However, we do not currently say what we actually mean by "identical", and it has been pointed out that defining this is not trivial. To resolve the issue for the MVP, we need to choose between the following options:

Option 1: Specify more precisely what we mean by "identical", possibly something like odr-identical modulo parameter names, return value identifier, and template parameter names, and make sure this is implementable.

Option 2: Specify that for two declarations of the same function in separate TUs, if both declarations have CCAs, that is always an ODR-violation (ill-formed, no diagnostic required), thus avoiding the issue. Note that this option does not seem practical because it would make any program IFNDR that includes the same header in different TUs if that header contains any function with a CCA.

Note that the original C++20 Contracts proposal allowed repeating CCAs in non-first declarations in the *same* TU if they were identical (see [P0542R5] section 2.7). However, this is not strictly necessary for the MVP and can be added post-MVP as an extension if needed.

2.2 Contracts on overridden and overriding functions

Status: Unresolved. Needs paper.

The current Contracts MVP, as recently amended by adopting [P2954R0], currently specifies that an overriding function shall not specify preconditions or postconditions, and inherits those of the overridden function. If a function overrides more than one function, neither the overriding function nor any of the overridden functions shall specify preconditions or postconditions.

Various people suggested that instead, we should go for a more conservative solution: neither the overriding function nor the overridden function shall specify preconditions or

postconditions, also in the single inheritance case, until we adopt a more comprehensive solution post-MVP. The concerns raised about the current approach include:

- Post-MVP, we want to have the option to either inherit or not inherit CCAs from an overridden function; "inherit" may be the wrong default here, so we should not bake that default into the MVP now;
- Having such a default may precludes widening the contract in a derived class;
- CCAs in overridden functions might be abused as a side channel to add functionality to overriding functions, in ways that Contracts are not intended to;
- If you consider a contract check to be part of a function's implementation, we effectively call a part of an overridden function's implementation from the overriding function, which some find surprising and undesirable;
- CCAs on overridden functions have the potential to break mixin systems that make use of virtual functions.

The above concerns are controversial and not universally shared in SG21, but nevertheless seem serious enough that we need a paper investigating this further. Our options are:

Option 1: Leave the MVP as it is.

Option 2: Tighten the MVP to say that no overriding or overridden functions shall specify preconditions or postconditions (which makes Contracts effectively incompatible with virtual functions).

Option 3: Relax the MVP by adding a more flexible solution for virtual functions, including in particular the ability to override inherited contracts.

Note that the original C++20 Contracts proposal allowed repeating CCAs on overriding functions, as long as they are identical, both for single and for multiple inheritance, but did not allow to override inherited contracts; this was so far always intended as a post-MVP extension.

2.3 Contracts on lambdas, and implicit captures

Status: Unresolved. [P2890R0] argues for one solution, [P2834R1] for another. Needs to be discussed and polled in SG21.

We need to specify whether CCAs should work on lambdas. There does not seem to be a reason why we should not allow this (see [P2890R0] for a more detailed discussion), and specifying this seems straightforward. However, there is one edge case with a controversy around the desired behaviour that needs to be resolved.

The current MVP specifies that all entities in a contract predicate are ODR-used, even if the CCA is unchecked (has ignore semantics). [P2834R1] explains why this must be so: ODR use is observable – it can trigger template instantiations and lambda capture – and we do not want the semantics of a CCA to affect the compile-time semantics of the surrounding code). In addition, since we adopted [P2877R0] for the Contracts MVP, the semantics of any

given CCA are not even known at compile time. The controversial case arises when a CCA on a lambda captures an entity not otherwise captured, and therefore the ODR-use of an entity in a CCA triggers an observable lambda capture?

```
auto f(int i) {
  return sizeof( [=] [[pre: i > 0]] {});
}
```

We have the following options:

Option 1: Allow this code. CCAs on lambdas follow the same rules for ODR-use triggering lambda captures as everywhere else in the language, and therefore the lambda will capture `i` and `f` will return `sizeof(int)`.

Option 2: Make this code ill-formed. CCAs on lambdas are not allowed to trigger captures of entities not otherwise captured.

Option 3: Do not allow CCAs on lambdas at all.

[P2890R0] argues for Option 1. This is most consistent with the rest of the language, least surprising to the user, and does not require introducing any new special rules around ODR-use and lambda captures. It is also consistent with attribute `[[assume]]` (see [P1774R8]), where we faced the exact same problem and resolved it in favour of Option 1.

[P2834R1] argues for Option 2. Allowing this code violates the principle of *zero overhead for ignored predicates* established in that paper. Such captures can cause an expensive copy of a captured object, or it can cause a lambda to no longer fit into the small object optimisation of `std::function`. It is therefore possible to construct a program where the mere presence of a CCA, even if its semantic is ignore, can cause runtime performance degradation, which is deemed a serious problem justifying making this case ill-formed.

Note that if, post MVP, we want to support calling functions from an unchecked CCA that do not have a definition (for example, checks like "does this pair of pointers specify a valid range" that are inexpressible in C++ but might be useful to a static analyser), we will need to relax the requirement that all entities in a contract predicate are ODR-used, at which point this whole discussion might have to be re-evaluated.

2.4 Contracts on coroutines

Status: Unresolved. Solution proposed in [P2957R0]. Needs to be discussed and polled in SG21.

We need to specify whether CCAs should work on coroutines. Coroutines are not like regular functions: a control can return to the caller even though the coroutine hasn't finished executing. This can raise some doubts whether the semantics of contract annotations as

defined for regular functions still apply. For the Contracts MVP, we need to choose between the following options:

Option 1: Allow preconditions, postconditions, and assertions in coroutines, and specify exactly what the semantics of preconditions and postconditions should be given the differences between coroutines and regular functions.

Option 2: Only allow assertions in coroutines, which are not affected by those differences.

Option 3: Do not allow any contracts in coroutines.

[P2957R0] argues for Option 1 and provides a set of possible semantics.

2.5 Contracts during constant evaluation

Status: Unresolved. Solution proposed in [P2894R0]. Needs to be discussed and polled in SG21.

We need to specify how CCAs should behave during constant evaluation. For CCAs that can be constant evaluated, and evaluate to `true`, the semantics are obvious: they should not have any effect whatsoever. The situation is less clear for the following two cases: CCAs that cannot be constant evaluated (because the predicate is not a core constant expression), and CCAs that can be constant evaluated but do not evaluate to `true`. For both cases, we need to choose from one of the following options for the Contracts MVP:

Option 1: Make the CCA ill-formed.

Option 2: Make the CCA ill-formed, no diagnostic required.

Option 3: Ignore the CCA during constant evaluation (modulo ODR-use).

Option 4: Ignore the CCA normatively, but recommend that a warning be issued.

We can pick a different option for each of the cases, and we can further make it implementation-defined whether one option or another should apply.

[P2834R1] contains some discussion on this matter and proposes a solution. However, the proposed solution is outdated: the paper ties the choice of compile time semantics to the given build mode (Option 1 if contracts is checked, and Option 3 otherwise). However, since we adopted [P2877R0] for the Contracts MVP, there are no more build modes, and the semantics of any given CCA (whether they are checked or not) are in general no longer known at compile time. The reasoning in [P2834R1] therefore no longer applies.

[P2894R0] contains a more up-to-date discussion of this problem space, and proposes to make it implementation-defined for both cases whether Option 1 or Option 3 applies.

2.6 Contracts on trivial special member functions

Status: Unresolved. Solution proposed in [P2834R1], needs to be discussed and polled in SG21.

When a trivial function, such as a trivial destructor, contains a contract annotation, does it affect its triviality? Clearly, the evaluation of a contract predicate is a non-trivial operation. For the Contracts MVP, we need to choose between the following options:

Option 1: An operation containing a CCA is never trivial.

Option 2: An operation containing a CCA may be trivial, however this may result in situations where the preconditions or postconditions are not checked due to the invocation of the special member function being skipped or replaced by a bitwise copy.

[P2834R1] discusses this in detail and recommends Option 2.

References

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

[P1774R8] Timur Doumler: Portable Assumptions. 2022-06-14

[P2521R4] Andrzej Krzemieński, Gašper Ažman, Joshua Berne, Bronek Kozicki, Ryan McDougall, and Caleb Sunstrum: Contract support — Record of SG21 consensus. 2023-06-15

[P2695R1] Timur Doumler and John Spicer: A proposed plan for Contracts in C++. 2023-02-09

[P2834R1] Joshua Berne and John Lakos: Semantic Stability Across Contract-Checking Build Modes. 2023-06-08

[P2877R0] Joshua Berne and Tom Honermann: Contract Build Modes, Semantics, and Implementation Strategies. 2023-06-13

[P2885R0] Timur Doumler, Gašper Ažman, Joshua Berne, Andrzej Krzemieński, and Ville Voutilainen: Requirements for a Contracts syntax. 2023-07-16

[P2890R0] Timur Doumler: Contracts on lambdas. 2023-08-10

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

[P2957R0] Andrzej Krzemieński: Contracts and coroutines. 2023-08-15

[P2894R0] Timur Doumler: Constant evaluation of Contracts. 2023-08-22