Contract Checks Should Be Regular C++

John Spicer (jhs@edg.com)

Document #: P3281R0
Date: 2024-05-15
Project: Programming Language C++
Audience: SG21, EWG

1. Introduction

SG21 has a “Contracts MVP” proposal [P2900R6] that intends to provide the “minimal viable product” for the first version of contracts to be added to the standard.

The intent of the MVP is to provide the core functionality needed by users without closing doors for future additions to the feature.

The Contracts MVP [P2900R6] makes a number of design decisions that prioritize certain use cases and/or design principals over others.

Some of these design decisions I believe will be surprising to many or most C++ programmers, both experienced C++ programmers and those new to the language.

The new semantics in [P2900R6] differ in fundamental ways from the C assert feature and from many or most user-written contract-like facilities.This will make it difficult for programmers to switch from existing facilities (either standard or user-written) to the facilities in the contracts MVP. There are issues, specifically with C assert that we should address, but I believe that the MVP or the proposed changes to the MVP would address those. The proposed changes would allow a migration path without (unneeded) changes in semantics.

2. MVP Semantics

2.1 MVP Semantics regarding duplication

The MVP goes into great detail to describe design motivations and consequences but is, in my opinion, surprisingly silent on the consequences of the design decisions with respect to side-effects. The concept of "destructive side effects" ([P2712R0], [P3228R1]) and its consequences for what you can and cannot do with the Contracts MVP is highly non-obvious and may be surprising to many developers. An example of what I would expect to do with contracts, but cannot with the MVP is:

```
struct X {};

/* Add x to a map and return true if it is not already in the map */
bool not_already_processing(X* x);

/* Remove x from the map and return true if it was found. */
bool done_processing(X* x);

void f(X* x) {
  contract_assert(not_already_processing(x));
  // Stuff that could possibly result in a recursive call
  // of f()
  contract_assert(done_processing(x));
}
```

This can fail under the MVP even in the absence of a recursive call because the checks could be elided in the recursive call or repeated in a non-recursive call.

2.2 Constification

The MVP makes variables with automatic storage duration implicitly const. This is done to prevent unintended modification of the parameters.

While I think this is a well-intended goal, it means that you can’t simply move code from (for example) an existing assert to a precondition or a contract_assert without potentially changing the meaning of the code.

This can include the selection of a different overloaded function. Some have asserted that such code uses a “bad overload set”, but I think the contracts feature is intended to provide a standard mechanism for what many (most) programmers already do, and constification is a barrier to such adoption. In addition, it is a “shallow” const, meaning that you can’t change the actual variable, but you can change things through (for example) a pointer.

```
void f(int* x) {
   pre(x = nullptr)  // error via MVP
   pre(*x = 1)       // okay via MVP
}
```

Constification can be worked around via a const_cast, but that requires programmers to already understand that the contract check has different semantics, and has bad ergonomics, is poorly understood by many programmers, and is discouraged by many coding guidelines.

3. The MVP vs. existing practice

The current language provides no explicit support for the duplication and elision semantics of the MVP. Consequently there is no existing code that is written with such semantics in mind. Conversely, all existing contract-like checking facilities such as C assert and user-defined assertion macros use the existing language, where the contract predicate is a regular C++ expression which is evaluated exactly once if the assertion is enabled. There is no way to easily gauge how many of the existing checks would work or fail given the new semantics, but all existing checks (modulo the discarded token aspect of C assert) should continue to work with the recommendations from this paper.

The EDG front-end makes use of many levels of sophisticated checking facilities, many of which require the maintenance of global state information. If we wanted to use the MVP facility, it would require extensive effort to review which checks could or could not be done using the new rules.

We would then be in a world where some checks were controlled by one set of flags and another set by a different set of flags, and we could not have the total set of checks controlled in the same way, nor could we have the checks share a contract-violation handling mechanism.

4. Recommendations

The MVP is intended as the starting point for Contracts in C++. As such, it should present a simple migration path from similar facilities such as C assert and user-written assertions based on the regular language semantics.

I believe that we need to have a Contracts facility that makes use of the regular C++ language to make it possible for users to easily migrate to the new facility.

We may also want a version of Contracts that supports the arbitrary elision and duplication of side-effects as described in the MVP, but I think that should be a post-MVP extension.

There have been arguments that such a facility cannot be added later, while the behavior recommended here could be, via some kind of opt-in syntax such as a label.

I think that whatever we choose now will require a language change later to allow the additional semantic, but I think future changes would be needed even without adding a different semantic.

The MVP does not allow for different contract behavior across different components of a project.

For example, if your product uses 10 external libraries, you might want to enforce checking only on your own code and not on the libraries. The MVP allows for such a scenario in principle, but does not explicitly provide any facilities to accomplish it or any kind of specification for how it would actually work (beyond "the selection of evaluation semantics is implementation-defined"), and I think that is appropriate for the MVP.

I believe that the long-term use of Contracts will require the support of independent contract scopes or domains that could have different rules for when they are enabled or disabled, and possibly even things like different contract-violation handlers.

The ability to have domain-dependent contract-violation handlers would provide a better long-term solution for those who want throwing contract-violation handlers and those who don't.

This would also facilitate the distribution of libraries in cases where a library vendor wants to ship only some versions of the library, which may not be possible with the MVP because things such as inline functions could potentially reference things that are not actually provided in the library if the client code is compiled in some other mode.

5. Proposed changes to P2900R6

For all three kinds of contract assertions (`pre`, `post`, and `contract_assert`), modify the following parts of the specification in [P2900R6]:

- **No *elision* of *checked* contract assertion evaluations:** In [basic.contract.eval], remove the allowance to determine and use the value of a predicate for performing a contract check without evaluating the predicate,
- **No *repetition* of *checked* contract assertion evaluations:** In [basic.contract.eval], remove the allowance to evaluate a previously evaluated contract assertion in a contract-assertion sequence again with the same or a different evaluation semantic,
- **No *constification* of contract predicates:** In [expr.prim.id.unqual], remove the rule that if an *unqualified-id* appears in the predicate of a contract assertion and the entity is a variable or structured binding with automatic storage duration or the result object of a function, `const` is is implicitly added to the type of the expression.

Acknowledgements

I would like to thank Timur Doumler very much for his feedback and his help with the wording.

Bibliography

[P2712R0] Joshua Berne (2022-11-13). Classification of Contract-Checking Predicates.

[P2900R6] Joshua Berne, Timur Doumler, and Andrzej Krzemieński (2024-02-29). Contracts for C++.

[P3228R1] Timur Doumler (2024-05-07). Revisiting side effects, elision, and duplication of contract predicate evaluations.