Contracts That Work

Document #:	P1429R1
Date:	2019-03-08
Project:	Programming Language C++
Audience:	EWG

Reply-to:	Joshua Berne <jberne4@bloomberg.net>
	John Lakos <jlakos@bloomberg.net>

Abstract

In a number of papers ([P1332R0], [P1333R0], [P1334R0], [P1429R0]) we have proposed a re-envisioning of how to structure the semantics of C++ contracts both to clarify their behavior and to enable solutions for important real-world use cases that come up immediately when attempting to introduce contracts into existing production codebases. Here, we aim to present refined wording for the fundamental changes we feel should be considered to solve the C++ community's needs and desires for contracts.

Contents

1 Revision history

- R1 – Additional wording clarifications
  - Consolidate to propose all semantics as explicitly usable.

  - Removed mention of disabling contract checking entirely.
  - Appendix sections A.1, A.2, A.3, and A.4.
- R0 – Initial draft

2 Overview

Contracts as originally specified in [P0542R5] consist of an attribute-like syntax for specifying contract-checking statements (CCSs) within the language. This syntax allows for specifying contracts as assertions (with `assert`), preconditions (with `expects`), or postconditions (with `ensures`). Each CCS contains a *conditional-expression* (or predicate) that is expected to be true when control flow reaches it. The current working paper (WP) fully captures how these different types of CCSs relate to code meaning, so we will focus only on how CCSs behave in relation to their surrounding code and other contracts by limiting our discussion to the `assert` "flavor" of CCSs.

The anticipated C++ contract checking facility (CCF) also introduces a *violation handler* that an implementation is allowed to let users define for themselves (more restrictive implementations are given the freedom to restrict the violation handler to one provided by the implementation). Our semantics seek to make behavior of a CCS independent of the violation handler. A violation handler may always choose to `throw` or `abort`, but to be the easiest to use for novices, we recommend that the default violation handler should "log" information (a stack trace, etc.) about the problem and return, leaving control flow up to the semantic in effect (for that specific CCS) at the call site.

What we seek to change is the way in which the behavior of a CCS is defined and determined, both in code and at translation time. That is what the rest of this paper will focus on.

3 CCS Semantics

There are 4 semantics that are essential to the definition of the C++ CCF. Here, we are merely giving names to the same semantics already accepted into the current draft of the working paper (WP) — all for the sake of improving clarity. As a brief aside, we reprise how history arrived at the semantics we are proposing here for C++.

- C's `assert` macro allowed for 2 fundamental behaviors — do nothing (compile out, or completely ignore) or abort on a failed check (which we call `check_never_continue`).
- Prior assertion facilities ([N3604], or Bloomberg's `BSLS_ASSERT`) often enabled pluggable behaviors for failed checks, including simply logging and continuing — we would call this behavior *check_maybe_continue*.
- With the adoption of contracts as a *language-level* facility ([P0542R5]) the option for an even more powerful way to leverage contracts became available by not checking contracts but allowing the compiler to treat a contract failure as language undefined behavior — a semantic which we would call *assume*. This was an option not achievable by any of the previous library-only solutions.

Putting concise, clear definitions of all possible CCS semantics into the WP for C++ 20 would improve future discussions of language interaction with contracts. Behavior can be discussed and defined in terms of the semantics instead of specifying the exact combination of translation options and code.

3.1 Ignore

The simplest semantic is to do nothing.

A CCS having the *ignore* semantic must be valid syntactically, but will otherwise never be evaluated — as if it was embedded in an unevaluated context.

3.2 Assume

A CCS having the *assume* semantic will be syntactically checked and may (might) be assumed (by the compiler) to be true. It is undefined behavior if the predicate were to be evaluated and it returned false. Notably, the expression is never evaluated (and thus functions used in the expression need not be defined for the program to be well-formed). Note also that there is no obligation on an implementation to actually do anything with this information, and a conforming implementation can freely treat this semantic identically to the *ignore* semantic.

3.3 Check (Never Continue)

A CCS having the *check_never_continue* semantic will evaluate the expression and if it is false will invoke the violation handler. If the violation handler returns, `std::abort` will be invoked. This guarantees that control flow will never continue if the predicate is false, so as a byproduct of that behavior the predicate can be known to be true after the CCS.

3.4 Check (Maybe Continue)

A CCS having the *check_maybe_continue* semantic will evaluate the expression, and if it is false, will invoke the violation handler. If the violation handler returns control flow continues as normal.

4 CCS Syntax

The current working paper allows each CCS to take an optional *level* (i.e., `default`, `audit`, or `axiom`), and if none is specified, `default` is assumed. A level is mapped to an actual semantic (e.g., *assume*, *check_never_continue*. That mapping is performed (for each TU) at build time.

[P1334R0] proposes a way to cut out the middle man and specify the intended semantic directly in the CCS itself. Explicit semantics like this bring us two useful properties:

- The CCS behavior is independent of build mode.

- The semantic of each CCS is independent of every other CCS in the same TU. It is this critically important property that allows staging a check that is new and unverified in the same TU as checks that are already fully enforced (or possibly even assumed). Note: The absolute requirement of having differing semantics for CCSs (e.g., even those on the same level) has been independently observed at larger software companies such as Google (e.g., by Richard Smith) and Bloomberg (e.g., by John Lakos).

Allowing an explicit `check_maybe_continue` CCS is a way to get a simple version of the "review" role discussed in-depth in [P1332R0]. A contract that is going to be a `default` level contract can be introduced first with the `check_maybe_continue` semantic, and it will then run "safely" without risking bringing down systems which previously were violating it in a "benign" fashion.

In the wording, where previously a *contract-attribute-specifier* contained an optional *contract-level*, now we would let that level be either a *contract-level* or a *contract-semantic*, and encapsulate that by defining a new grammar non-terminal *contract-mode*. The allowed explicit semantics all need to be added to **[lex.name]/2** — identifiers with special meaning — as well as the grammar for *contract-semantic*.

5 Formal Wording

In **[gram.dcl]** the following is changed:

contract-attribute-specifier
`[ [ expects` ~~*opt contract-level*~~ *opt contract-mode* `:` *conditional-expression* `] ]`
`[ [ ensures` ~~*opt contract-level*~~ *opt contract-mode* *opt identifier* `:` *conditional-expression*
`] ]`
`[ [ assert` ~~*opt contract-level*~~ *opt contract-mode*`:` *conditional-expression* `] ]`

contract-mode
contract-level
contract-semantic

contract-semantic
`check_maybe_continue`
`check_never_continue`
`ignore`
`assume`

In **[lex.name]**, four identifiers, `check_maybe_continue`, `check_never_continue`, `ignore` and `assume`, are added to the table Identifiers with special meaning.

[dcl.attr.contract.syn]/6 gets the following changes:

The only side effects of a *checked* predicate that are allowed in a *contract-attribute-specifier* are modifications of non-volatile objects whose lifetime began and ended within the evaluation of the predicate, or invocation of the *violation handler* and any side effects that function might have. An evaluation of a predicate that exits via an exception invokes the function `std::terminate` The behavior of any other side effect is undefined.

[dcl.attr.contract.check] gets replaced by the following (3 paragraphs removed completely for brevity):

If the *contract-* ~~*level*~~ *mode* of a contract-attribute-specifier is absent, it is assumed to be a *contract-level* of `default`. [*Note:* A `default` *contract-level* is expected to be used for those contracts where the cost of run-time checking is assumed to be small (or at least not expensive) compared to the cost of executing the function. An `audit` *contract-level* is expected to be used for those contracts where the cost of run-time checking is assumed to be large (or at least significant) compared to the cost of executing the function. An `axiom` *contract-level* is expected to be used for those contracts that ~~are formal comments and are not evaluated at run-time~~ cannot be checked at run-time - i.e., they cannot be implemented or would require side-effects to execute. A *contract-semantic* is expected to be used for those contracts that need a semantic independent of the *build mode*. — *end note*]

The *violation handler* of a program is a function of type "*opt*`noexcept` function of (lvalue reference to `const std::contract_violation`) returning `void`". The violation handler is invoked ~~when the predicate of a checked contract evaluates to `false`~~ when the semantic given to a contract indicates it should be (called a *contract violation*). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the `std::contract_violation` argument value is set, except as specified below. [*Note:* Implementations are encouraged but not required to provide a default *violation handler* that outputs the contents of the `std::contract_violation` object then returns normally. — *end note*] If a precondition is violated, the source location of the violation is implementation-defined [*Note:* Implementations are encouraged but not required to report the caller site. — *end note*] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

If a violation handler exits ... (unchanged note on throwing violation handlers and `noexcept` functions.)

Every contract will be given a semantic at translation time that is *ignore*, *assume*, *check_never_continue*, or *check_maybe_continue*.

A translation may be performed with varying *build modes*. The mechanism for selecting *build modes* is implementation-defined. The translation of a program consisting of translation units where the *build mode* is not the same in all translation units is conditionally-supported, with implementation defined semantics. There should be no programmatic way of setting modifying, or querying any part of the *build mode* of a translation unit. The build mode contains the following values:

- An *axiom contract mode* with a value of *assume* or *ignore*. If specified, all *axiom* level contracts have the named semantic.
- A *default contract mode* with a value of *assume*, *ignore*, *check_maybe_continue* or *check_never_continue*. If specified, all *default* level contracts have the named semantic.
- A *audit contract mode* with a value of *assume*, *ignore*, *check_maybe_continue* or *check_never_continue*. If specified, all *audit* level contracts have the named semantic.

If not specified by the *build mode*, a `default` level contract has the semantic *check_never_continue*.

If not specified by the *build mode*, an `audit` level contract has the semantic *ignore*.

If not specified by the *build mode*, an `axiom` level contract has the semantic *ignore*.

A contract having a *contract-semantic* has the semantic named by that *contract-semantic*.

A contract having the semantic *ignore* is not checked and not evaluated.

A contract having the semantic *assume* is not checked and not evaluated. It is undefined behavior if the predicate of such a contract would evaluate to false. [*Note:* The implementation may freely evaluate any parts of the predicate that are available to it which do not have side effects, thus allowing it to "prove" the violation of the predicate. — *end note*]

A contract having the semantic *check_never_continue* is checked. The predicate is evaluated and if it returns false the violation handler is invoked. If the violation handler returns normally, `std::abort` will be invoked.

A contract having the semantic *check_maybe_continue* is checked. The predicate is evaluated and if it returns false the violation handler is invoked.

The predicate of a contract that does not have a checked semantic in any *build mode* is an *unevaluated operand*.

During constant expression evaluation (7.7), any call to the violation handler is ill-formed. [*Note:* Contracts with checked semantics are still evaluated as normal, thus any *contract violation* of a checked contract during constant expression evaluation will be ill-formed. — *end note*]

During constant expression evaluation, a violation of a contract having the *assume* semantic is ill-formed, no diagnostic required. [*Note:* Implementations are encouraged to fail to translate any compile-time contract violations they identify, but are not required to identify all such violations when they might involve undefined functions, side effects, or infeasible runtime complexities. — *end note*]

6 Conclusion

These proposed wordings capture the original stated intent of the merged contract proposal, with the addition of the features in [P1333R0] and [P1334R0].

7 References

[N4800] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4800.pdf

[N3604] John Lakos, Alexei Zakharov, *Centralized Defensive-Programming Support for Narrow Contracts*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3604.pdf

[N4075] John Lakos, Alexei Zakharov, Alexander Beels, *Centralized Defensive-Programming Support for Narrow Contracts (Revision 6)*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4075.pdf

[P0542R5] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup, *Support for contract based programming in C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html

[P1290R0] J. Daniel Garcia, *Avoiding undefined behavior in contracts*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1290r0.pdf

[P1290R1] J. Daniel Garcia, Ville Voutilainen *Avoiding undefined behavior in contracts*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1290r1.pdf

[P1290R3] J. Daniel Garcia, *Avoiding undefined behavior in contracts*

[P1332R0] Joshua Berne, Nathan Burgers, Hyman Rosen, John Lakos, *Contract Checking in C++: A (long-term) Road Map*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1332r0.txt

[P1333R0] Joshua Berne, John Lakos, *Assigning Concrete Semantics to Contract-Checking Levels at Compile Time*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1333r0.txt

[P1334R0] Joshua Berne, John Lakos, *Specifying Concrete Semantics Directly in Contract-Checking Statements*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1334r0.txt

[P1335R0] John Lakos, *"Avoiding undefined behavior in contracts" [P1290R0] Explained*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1335r0.txt

[AKRZEMI1] Andrzej Krzemieński, *Assigning semantics to different Contract Checking Statements*
https://github.com/akrzemi1/__sandbox__/blob/master/papers/ccs_roles.md

[P1429R0] Joshua Berne, John Lakos *Contracts That Work*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1429r0.pdf

A Other Wording Improvements

Changes discussed in this section have been incorporated into the formal wording above, and we see them as bug fixes on the previous version of this document and the working paper. These are the result of further refinement, reflector discussions, and discussion with a number of people at the 2019 Kona meeting.

A.1 constexpr Evaluation

Wording has been added to explicitly call out how contract violations should be dealt with during constant expression evaluation. The violation handler users might install is intended to be part of the system they deploy, and requiring that to be `constexpr` evaluatable seems not to be helpful. In order to catch checks at compile time, making a call to the violation handler ill-formed, diagnostic required, achieves this goal.

For assumed contracts, we do not want to require that the compiler actually try to evaluate the expression, but we do not want to preclude compilation failures when a failure is noticed by the compiler. Because of this, we make violations of an assumed contract ill-formed, *no* diagnostic required.

A.2 terminate or abort

We have chosen `std::abort` for now on the advice of other committee members.

- Arguments for `std::terminate`
  - There are no cases where the language currently injects any direct calls to `std::abort`.
  - The use of `std::terminate` outside of exception handling has arguable already happened.
- Arguments for `std::abort`
  - Contract violations should not call and confuse custom `terminate` handlers, especially since the violation handler is already configurable.
  - `std::terminate` as designed was for failures in the exception handling system, which contract violations are not.
  - Failure to use `std::abort` outside of exceptions before should not be continued now.

A.3 Side effects of predicates

The side effects of unchecked predicates being undefined was not a part of initial proposals, and having predicates with side effects be usable for `axiom`-level contracts has some apparent use cases. Because of that we should also be changing **[dcl.attr.contract.syn]/6** to be limited only to checked contracts. Similarly, a predicate that itself violates a checked contract should not be treated as undefined behavior, so that should be added to the list of allowed side effects (on the assumption that the contents of most useful violation handlers will contain side effects of one form or other).

It should be considered for the future that we narrow the undefined behavior of side effects to wording that simply leaves the evaluation of the predicate unspecified, but the specifics of that change can be left to future revisions of the standard. By leaving the majority of checked predicates with side effects as undefined behavior we retain the ability to narrow that undefined behavior in any way desired in the future.

A.4 Axiom Behavior

Axioms as originally advertised allowed for 2 features that have not been successfully transcribed into the wording for the standard.

- An axiom should be able to reference functions that are declared but not defined in order to reference verbs meaningful only to static analyzers or to the compiler at compile time. Currently the compiler may always evaluate any predicate (checked or not) ad this prevents that.
- An axiom should be able to reference functions with side effects and trust they will not evaluate. All predicates with side effects are thus labeled undefined behavior. This includes predicates that are "never executed at runtime" — i.e., axioms.

The previous section addresses the second issue. To fix the first issue, after discussing the details with core, we have modified how "assumed" contracts are worded, and have added the statement that any contract that cannot be checked in any build mode is a *unevaluated operand*. This maintains the feature that if a predicate *can* be checked the build mode you choose does not impact what is ODR used or not.