Contracts That Work

Document #:	P1429R2
Date:	2019-06-16
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

- R2 – Refinements (Pre-Cologne, June 2019)

  - Allowing side effects in all predicates
  - Allowing checked predicate elision
- R1 – Additional wording clarifications (Post-Kona, February 2019)
  - 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 (Pre-Kona, January 2019)

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 — [N4810]) 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 — 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 a valid expression - similar to the arguments of a `sizeof` operator, the CCS predicate is a discarded statement. The actual predicate will not, however, ever be evaluated and the CCS will have no other impact on the program meaning.

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 return false. Notably, the expression is never actually 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 might evaluate the expression and if would be 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 might evaluate the expression, and if it would be 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).

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 Build Modes

Contract control at build time is an essential part of a complete CCF. Being able to deploy the same code in modes where the speed versus safety choice is made with different priorities facilitates robustly deploying highly optmized and safe software.

CCSs with levels (`default`, `audit`, or `axiom`) should be independantly controlled (see [P1332R0] for a more thorough discussion of why). In addition, numerous parties have expressed a strong need to be able to turn off all CCSs completely (both to be absolutely sure program behavoir is independant of predicate executiona and to obtain a build that never invokes the violation handler or has arbitrary undefined behavior.) To facilitate these needs, we define a *build mode* that contains a number of different values which together determine which semantic any given contract might have.

6 Formal Wording

In **[basic.def.odr]/12.6**:

- ...

- if `D` invokes a function with a precondition, or is a function that contains an assertion or has a contract condition (9.11.4), it is implementation-defined under which conditions all definitions of **D** shall be translated using the same ~~build level and violation continuation mode~~ build mode; and
- ...

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 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.~~ 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 have significant side-effects if evaluated. 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 if indicated by the semantics of the contract when the predicate of a ~~a checked~~ contract ~~evaluates~~ would evaluate

to `false` (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 (checked in order if specified to determine the semantic of any given contract):

- A *global contract mode* with a value of *enabled* or *disabled*. If specified as *disabled*, all contracts have the *ignore* semantic. [*Note:* A *global contract mode* of *enabled* has no effect. — *end note*]
- 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*.

If not specified by the *build mode*, 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; if such a contract would evaluate to false during constant expression evaluation it is unspecified if that expression is a core constant expression; if the predicate of such a contract would evaluate to false in other contexts, the behavior is undefined.

A contract having the semantic *check_never_continue* is checked, it is unspecified if the predicate is evaluated. If the predicate of such a contract would evaluate to false, the violation handler is invoked. If the violation handler is invoked and returns normally, `std::abort` will be invoked.

A contract having the semantic *check_maybe_continue* is checked, it is unspecified if the predicate ise valuated. If the predicate of such a contract would evaluate to false, the violation handler is invoked.

When the return value of the predicate of a contract with a checked semantic can be determined without evaluating that predicate, an implementation is allowed to omit the evaluation of the predicate, even if the predicate has side effects. [*Note:* Side effects of predicates are discouraged as the validity of a program should not depend upon the evaluation (or not) of contract predicates. — *end note*]

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

[*Note:* During constant expression evaluation (7.7), an expression is not a core constant expression if it invokes the violation handler, and it is unspecified if it is a core constant expression if it violates an assumed contract. — *end note*]

[expr.const]/4 gets the following changes: An *expression* `e` is a core constant expression unless the evaluation of `e`, following the rules of the abstract machine (6.8.1), would evaluate one of the following expressions:

- ...
- ~~a checked contract (9.11.4) whose predicate evaluates to false;~~ any invocation of the *violation handler*;
- ...

If `e` satisfies the constraints of a core constant expression, but evaluation of `e` would evaluate an operation that has undefined behavior as specified in Clause 16 through Clause 32 of this document, a contract with the *assume* semantic (9.11.4) whose predicate would evaluate to false, or an invocation of the `va_start` macro, it is unspecified whether `e` is a core constant expression.

7 Conclusion

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

8 References

[N4810] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n4810.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

[P1607R0] Joshua Berne, Jeff Snyder, Ryan McDougall http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1607r0.pdf

[P1670R0] Alisdair Meredith, Joshua Berne *Side Effects of Checked Contracts and Predicate Elision*

[P1671R0] Joshua Berne, Alisdair Meredith *Contract Evaluation in Constant Expressions*

[P1704R0] Andrzej Krzemieński, Joshua Berne *Undefined functions in axiom-level contract statements*

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. Many also have separate proposals that look to add these features independantly of this proposal.

A.1 constexpr Evaluation

See [P1671R0] for a discussion of this issue independant of the more foundational issues in this proposal. This proposal incorporates the ideas and wording presented there.

Wording has been added to explicitly call out how both checked and unchecked contract violations should be dealt with during constant expression evaluation. A checked contract violation was already not-a-constant-expression, so any attempt to force compile time resolution of a function with a checked contract violation would fail. An unchecked contract was previously always valid, and we propose

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

See [P1670R0] for a discussion of this issue independant of the more foundational issues in this proposal. This proposal incorporates the ideas and wording presented there.

The current draft declares almost all side effects of checked contracts to be undefined behavior. For unchecked contracts (`axioms`) with the associated draft wording that allows evaluation this removes the ability to safely use any contract that has a side effect. For checked contracts, this makes a wide variety of functions that conceptually have no side effect become introducers of undefined behavior -

this includes any predicate that might need to allocate, log, access volatile variables, or use any c function that might set `errno`. Precluding that entire class of functions from safe use in predicates is problematic.

The desire to not allow side effects mostly stems from a desire to maintain the optionally-executable nature of contracts. If a function has no side effects, you have no actual way to observe if it was evaluated or not, so a compiler can freely elide the evaluation if it can determine the result without executing it. Similarly, if the function has no side effects then correct programs that behave observably differently in different build modes are precluded.

If we have checked semantics always evaluate contracts then the side of those predicates become something users can depend upon. In order to prevent this, we explicitly allow the implementation to omit the evaluation of the predicate if it can determine what the return value of the predicate would be. Since the implementation still needs to determine if the predicate returns true or false, compilers are left with two choices - evaluate the predicate or evaluate something without side effects that provably always returns the same value as the predicate. This gives compilers with enough semantic analysis the ability to freely elide contracts that it can prove are not being violated, while not turning 'benign' side effects into dangerous pitfalls of undefined behavior.

A.4 Axiom Behavior

See [P1704R0] for a discussion of this issue independant of the more foundational issues in this proposal. This proposal incorporates the ideas and wording presented there.

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 an *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.

A.5 Global Contract Mode

There is an inherent tension in terms of teachability and consistency with the introduction of concrete semantics.

- Some people have expressed a strong desire that anything that looks like a contract should have no functional impact on a program's behavior, so all contract attributes should support being "turned off".
- Others have expressed a desire that when explicitly stating as semantic that line of code should actually have that stated semantic, making it easier to test, build alternate facilities, and reliably know in limitted cases exactly how a specific piece of a program will behave.
- Others still see the benefits of the concrete semantics on their own (especially `assume`), and feel they would lose those benefits if there was an underlying off switch for those statements, so if given a solution where the literal semantics were not actually literal we would then see proposals for redundant facilities that spell almost the same behavior in a totally different fashion.

We have put in a *global contract mode* which, when *off*, will force all CCSs to have a semantic of *ignore*. The existence of this mode needs only one line of wording, and we think a simple vote on the desire for it should determine the inclusion of that line in the standard.

A.6 Minimizing Contracts

One additional suggestion that has been made, [P1607R0], was to remove the build-time controls and levels almost entirely. That proposal followed the wording in this paper very closely, simply removing `audit`, `axiom`, and everything in the *build mode* other than a single on/off switch.

That approach remains a subset of the wording in this document, and would be easily applied if that route was deemed preferable.