Contracts That Work

Document #:	P1429R0
Date:	2019-01-22
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]) 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 *only* the fundamental changes (taken from those proposals) that are absolutely essential to making contracts work properly in as concise and direct a way as possible.

Contents

1 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 introduce 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 to call `std::terminate`, 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.

2 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 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 call *assume*. This was an option not achievable by any of the previous library-only solutions.

- The nuances related to how continuing after a violation handler should be treated are complicated and subtle. We discuss details of how `check_maybe_continue` might be improved and another semantic, `check_always_continue` in appendix A.

Putting concise, clear definitions of the universe of possible CCS semantics into the WP for C++20 would improve future discussions of how other features of the language interact with contracts. Behavior can be discussed and defined in terms of the semantics directly instead of always having to specify the exact combination of translation options and code needed to indicate the behavior being discussed.

By adopting this terminology directly in the standard, we obviate copious redefinition and convoluted explanation in terms of arcane/baroque compiler-flags and build modes.

2.1 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.

2.2 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.

2.3 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::terminate` 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.

2.4 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.

The specifics of what the compiler is (and is not) allowed to conclude after a violation handler returns normally can be very subtle and allow for a wide variety of different expectations with different trade-offs. We will go into details of how this definition might be further refined and discuss an additional, subtly different semantic, *check_always_continue*, in appendix A.

3 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 serves a place holder for the actual semantic (e.g., *ignore*, *check_never_continue*, *check_maybe_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 (almost — see section 6) 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 semantic, `check_maybe_continue`, needs to be added to **[lex.name]/2** — identifiers with special meaning.

4 Assigning Semantics to Levels (At Build Time)

The current WP provides a *build level* and *violation continuation mode* that together control which semantics each level gets in a particular TU.

We propose a vastly simpler and more powerful direct approach in which each level (`default`, `audit`, or `axiom` can be assigned (independently) any semantics valid for that level. Although the current WP does not allow `axiom` to be changed from it's default *assume* semantic, we propose that `axiom` level be allowed to take on exactly two semantics: *assume* (by default) and *ignore*.

Section 4.1 of [P1332R0] goes into extensive consideration of which of the 32 (or 50 with the addition of the fifth semantic) possible semantics are 'safe' and worth allowing, and more reasons to include levels not justified there have been discussed elsewhere. A restrictive subset could be adopted, but to provide freedom to do what users might want to do we suggest allowing all such mappings (with possible future notes to warn against permutations that might be highly likely to lead to misuse).

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
explicit-contract-semantic

explicit-contract-semantic
`check_maybe_continue`

In **[lex.name]**, one identifier, `check_maybe_continue`, is added to the table Identifiers with special meaning.

[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. — *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. 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 an assignment of *default* and *audit* to any available semantic and *axiom* to any non-runtime-checked semantic. If unspecified, *default* should be assigned to

check_never_continue, *audit* should be assigned to *ignore*, and *axiom* should be assigned to *assume*. A contract with a *contract-level* will be given the semantic assigned to that level. There should be no programmatic way of setting, modifying, or querying the assignment of semantics to levels. The translation of a program consisting of translation units with different assignments of levels to semantics is conditionally-supported.

A contract with an *explicit-contract-semantic* will be given the named semantic.

A contract having the semantic *ignore* is not checked and not evaluated. [*Note:* Note that the predicate must still be syntactically valid as stated elsewhere. — *end note*]

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:* It is not required that the predicate is actually evaluatable — the predicate may reference (for documentation purposes) functions that are declared but never actually defined, and the implementation may not evaluate the function and thus require the definition of those functions — *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::terminate` 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.

6 Disabling Contract Checking Entirely

Part of the primary pedagogical basis of contract checking is that contract checks are not part of a function's implementation and should always be considered optional. As a way to keep this rule, we would add a single build option — *contracts mode* — which defaults to *on* but when *off* causes *all* contracts to have the *ignore* semantic.

This would cause any alternate CCF built on top of explicit semantics to be subject to this same overriding flag — but in exchange all CCFs that might be created would remain "pure" contract handling facilities.

6.1 Additional Formal Wording

This feature would need another paragraph added to **[dcl.attr.contract.check]** to say the following:

A translation may be performed with one of the following *contract modes*: *on* or *off*. If the *contract mode* is *off* then all contracts have the semantic *ignore*. This takes precedence over all other ways in which semantics might be given to a contract. If unspecified, the *contract mode* is *on*.

7 Making All Semantics Available Explicitly

While allowing an explicit *check_maybe_continue* CCS is imminently useful, once the semantics are defined it seems pointlessly restrictive to not make them available as well, and the benefits of doing will lead to long-term results as we get industry-wide experience with more widely applicable mature CCFs.

Allowing explicit `ignore` is an obvious extension that would allow staging of new contracts, or total disabling of questionable contracts without just commenting them out and leaving them susceptible to code rot.

Allowing explicit `assume` would open up the whole world of custom optimizations in a completely opt-in fashion. Any group using a macro like facility to choose semantics would be able to take their best tested production code and optimize it for exactly what it is doing — treating any contract violations as undefined behavior — potentially seeing huge performance improvements with minimal risk (as the contracts in question will have already been run extensively with a checked semantic).

Allowing explicit `check_never_continue` CCSs would open up the door for people to build alternate level-like facilities (admittedly using macros). These could provide for arbitrary numbers of roles and levels as in [P1332R0], or capture the needed workflows associated with other developer intentions like those discussed in [AKRZEMI1].

All of the macro-based facilities that could sit on top of explicit semantics will bring real, extensive industry experience to the proposals for the next standard so that language level contracts can evolve into what people are actually going to use. Though we see macros as the only way to tie together this feature for now, getting this experience so we can build a more flexible system into the language later will give us a way to obviate any need for macros for contracts.

7.1 Additional Formal Wording

The wording change to add additional explicit semantics is almost entirely covered by section 5 — Each additional semantic only needs to be added to **[lex.name]/2** as an identifier with special meaning and to the production for *explicit-contract-semantic*.

8 Conclusion

We believe that rewording in terms of named semantics (section 2), allowing at least `check_maybe_continue` as an explicit semantic (section 3) and allowing free assignment of semantics to levels (section 4) are essential to making use of contracts at scale in the coming years post-C++20 and pre-(C++20)++.

Allowing all of the semantics to be stated explicitly allows those facilities to be more robust and interact with the builtin CCF significantly more cleanly (by allowing those facilities to choose any semantic explicitly) and makes the entire feature seem like a complete, well-rounded part of the language. Our strong recommendation is to also adopt section 7.

The *contract-mode* to control all contracts (with a or with an explicit semantic) keeps the feature more pedagogically sound — i.e., whether or not contract checking is enabled should — by definition — have no effect on the meaning (behavior) of correctly written programs. This flag also removes the ability to write a CCS and *know* what that statement will do in any build mode, which many more pragmatic programmers might find desirable. We leave it up to the committee at large to decide which of these choices is preferred.

9 References

References

[N4727] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4727.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

[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

A Continuing After Violation

It can be quite surprising about how much impact the code after a CCS can have on a contract itself. Part of the problem is that often CCS predicates explicitly check against the things that are going to lead to *language* undefined behavior (UB) if they are violated.

Continuing after a violation then leads control flow directly into that UB. This brings all the scary voodoo to the table and at best elides your CCS entirely, and more likely brings the nasal demons to your door.

If the compiler knows the violation handler (i.e. when doing link-time optimization) and knows it will always return (such as a violation handler that simply calls `printf`) then UB after the contract check freely travels back in time to elide the check itself. If the compiler does not know the violation handler, or cannot determine that it will always return, then the violation handler itself provides a barrier against that time travel.

[P1332R0] identifies these two different conclusions based on the violation handler as two different semantics you might want to specify in different situations independently of the actual installed violation handler or how much the compiler knows about it.

- *check_maybe_continue* – The compiler is told not to know whether the violation handler returns normally or not.
- *check_always_continue* – The compiler is told to treat the violation handler as if it was a function that does always return.

So if you are trying to debug a program, you'd like to know that when you insert an assertion that it actually works and isn't elided by the very bug you were trying to uncover, right? Well when you're trying to develop code at your desk, you want to use *check_maybe_continue* so that you have the best shot at it's not being optimized away.

On the other hand, if you are running code in production and you want to add a new check, the last thing you want to do is crash the system because you optimized some code, and allowed a fatal bug, that was previously elided to surface. It is precisely those cases where you would deliberately choose the *check_always_continue* over the *check_maybe_continue* if such were available.

A.1 The Scary Case for Check (Maybe Continue)

Consider the following code with any contract semantic for the `assert` that allows control to continue after the *violation handler* is invoked:

```
int foo(int *p) {
  [[assert : p ]];
  return *p;
}
```

With link time optimization and a violation handler that just calls `printf`, this behaves as-if it were the following code:

```
int foo(int *p) {
  if (!p) {
    printf("OOPS");
    // nothing special here
  }
  return *p;
}
```

With nothing other than this code the compiler can know (because `printf` is a function the compiler fully understands which will always return normally) that control flow will return the `*p` regardless of what the value of `p` is. This means that `p` being `nullptr` would lead to undefined behavior, so the entire `if` block can freely be elided, including any logging of the contract violation.

The *check_maybe_continue* semantic is one solution to this pressing problem. [P1332R0] proposes that this semantic include a requirement that the compiler never assumes that the violation handler returns or does not return. This assumption should prevent the knowledge that `p` is not null at the `return` statement from "time traveling" backwards to elide the contract check.

A.2 The Scary Case for Check (Always Continue)

Consider this piece of legacy code, and the poor developer tasked with trying to make it more robust by adding CCSs to it.

```
  inline void eat_them(T *p)
  {
    if (!p) { eat_all_kittens(); }
  }
  inline void save_them(T *p)
    //[[expects: p]]
  {
    p->save_the_kittens();
  }
  void foo(T *p)
  {
    eat_them(p);
    save_them(p);
  }
  // somewhere in a far off file that does not inline foo...
  void bar()
  {
    foo(nullptr);
  }
```

With that seemingly innocuous CCS commented out, the same logic that motivated *check_maybe_continue* would elide the check and keep the kittens safe. A brave developer seeking only to make it safer to keep kittens safe uncommenting the CCS — if it had any semantic that did not allow continuing to the dereference of `p` or did not allow the knowledge of `p` being dereferencable to

propagate backwards to the first check — would suddenly force the call to `eat_them` to not be elided. The poor caller of `bar`, previously unaware that their code had any problems (perhaps because this was all written in the dark, kitten-hating days of the previous century) would now find themselves with the blood of many cute furry creatures on their hands.

This seems contrived, but who among us has not seem similarly horribly and subtly broken code moving along without issue until an unrelated change like this is introduced to bring a pile of extra darkness into the world? This semantic is not for every day use, but in large organizations that want to introduce contracts into extensive and currently immeasurably broken systems that currently "work fine" this semantic provides the minimal first step every injected contract can take on its way to being fully enforced (with *check_never_continue* or leveraged for optimization with *assume*.