Requirements for Contract Roles

Document #:	P1606R0
Date:	2019-03-08
Project:	Programming Language C++
Audience:	EWG

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

Abstract

The introduction of contracts to the C++ language is going to lead to the deployment of contract checking statements (CCSs) in every layer of the software development stack - from the standard library up to the applications that are exposed to users. When deciding what semantic each CCS should have it is important both to understand the tradeoffs involved and what factors should be considered. It is equally important that in the long term C++ provides a way to facilitate making those decisions correctly, and highly preferable that the decisions are easily made in an acceptable way when using contracts in C++ 20.

This paper provides a discussion of the various inputs into the decision on CCS semantic and how that might be implemented with the currently proposed working draft, explicit semantics as introduced in [P1429R0], or builtin support for roles similar to [P1332R0].

Contents

1 Choice of Semantics **1**
- 1.1 Difficulty of Checking . . . 2
- 1.2 Code Covered By a CCS . . . 2
- 1.3 Age of a CCS . . . 3
- 1.4 Type of Deployment . . . 3

2 Solutions **4**
- 2.1 Macros on [N4800] . . . 5
- 2.2 Macros on [P1429R0] . . . 6
- 2.3 Roles . . . 7

3 Conclusion **8**

4 References **8**

1 Choice of Semantics

[P1429R0] proposed four semantics for CCSs, and we will reference those. In general, these are the semantics and where they might be most applicable to be used:

- ignore — Checks that have not been verified but are too expensive in the current deployment to execute.
- check_maybe_continue — Checks that are being introduced for the first time into already stable code.
- check_never_continue — Checks that are cheaper to check than the cost of their failure would be, providing defensive protection against what might happen should they be silently broken.
- assume — Checks where there is both confidence in the current deployment that they will not fail *and* where there is an advantage for generated code to leverage that assumption.

The important points for deciding on these categories revolve around the cost of the check, the stability of the code being checked, the age of the check, and for what specific purpose that build is being deployed.

1.1 Difficulty of Checking

The current draft ([N4800]) identify this as the only metadata thing that can be captured at code-writing time about a CCS by specifying a *contract level* of *default*, *audit* or *axiom*. This information about the cost or possibility of checking at runtime is one of the largest drivers of which semantics might be appropriate.

"Uncheckable" predicates — those which require a side effect or reference "magic" functions that do not have actual implementations — simply cannot be assigned any evaluated semantic. They are restricted to just the *ignore* and *assume* semantics.

Predicates which could be checked at runtime (their implementations exist and have no side effects) can then be broken down by how much they might cost to implement. Existing proposals suggest *audit* for "slow" predicates and *default* for "fast" predicates.

Checks that break a function's complexity guarantee are obviously in the "slow" category. The remaining checks, however, are very dependent on the performance sensitivity of the users of a function. Some checks will be essentially free — validating values that are already in cache and doing very simple comparisons, resulting in a jump easily marked as unlikely to take. Some checks will be subjectively expensive - requiring work comparable to or even many times more expensive than the function invocation itself, touching of external resources, or other costs that might drastically pessimize a function if they were evaluated.

1.2 Code Covered By a CCS

Deciding if a CCS should be "enforced" with a *check_never_continue* semantic, "monitored" with a *check_maybe_continue* semantic, or even *assumed*, is in many ways a function of the age and stability of the CCS and the code that the CCS is checking.

Which code is checked by a CCS depends on type of check that it is.

Preconditions generally detect bugs in the code calling a function. Importantly, preconditions only check a subset of the full contract of a function (ideally, a non-proper subset, but often parts of a function's English preconditions cannot be checked in code).

Asserts within a function check that the body of that function performs correctly given the preconditions that were checked. At times an assert can also check parts of a precondition that could not be completely checked at calling time (consider verifying that a looked-up value has the correct state without requiring doubling the lookup in the precondition check, or checking that part of an array is sorted as you navigate it in a binary search without doing a full $O(n)$ scan of the array).

Postconditions, similarly to asserts, check that the function behaved properly in cases where a precondition was checked. It is also possible for postconditions to catch an unchecked part of an English language precondition.

1.3 Age of a CCS

The age of a CCS also strongly impacts how safe it is to enforce it or assume it. Obviously, a newly added CCS in already existing code puts users at risk of the CCS check itself being improperly stated. This is especially common when boundary conditions might be inconsistently depended on by an implementation, or poorly documented in an English language contract.

An important case to consider, however, is any situation where CCSs can be changed at build time between being ignored and being checked. Deploying a build to a system that previously had a CCS ignored is functionally identical to writing the CCS for the first time, without any need to alter that code.

In both of these cases, if a CCS is the same age as the code it checks it is generally best to enforce the CCS with the *check_never_continue* semantic so that misuses are caught quickly and bad results do not get acted upon. When a CCS is newer than the code that it checks, it is important to have a facility to enable it to identify if it is being broken without the associated cost of aborting if the CCS is, in fact, being violated. This is the primary use for the *check_maybe_continue* semantic. Only once a CCS has been monitored for long enough that one can conclude the risk of aborting is low enough for the existing deployment should *assume* be considered as a valid semantic.

1.4 Type of Deployment

Software gets built and deployed in many different ways as it goes through its life cycle from development to production use. At each stage of this life cycle, contract checks should be leveraged in different ways with different semantics.

When developing code locally, any CCSs that might be broken by the code you are writing should be enabled and checked. This can often mean wanting to turn on some or all "slow" checks to validate behavior. In general, all such checks should be in the *check_never_continue* semantic or, if the cost of the checks is too large or the check is too remote from the code being changed, the *ignore* semantic. The continuing *check_maybe_continue* is far less useful during development as it

is less likely that such violations will be easily noticed and acted upon when writing and running new code.

Unit tests should similarly often built to fail fast on any contract violation that can be speedily enough detected. For thoroughness, one might also want to validate that a system passes all unit tests with all CCSs *ignored* and/or *assumed*. The *check_maybe_continue* semantic is again less useful in this environment.

Shared deployments environments where performance is not critical but correctness of newly deployed code is being validated most actively by human beings will often want to enable as many checks as possible with the *check_never_continue* semantic. Often, though, new CCSs in this environment that are being frequently violated would be too disruptive on unrelated work, and the *check_maybe_continue* semantic if its output is properly monitored can be very useful for introducing new checks into existing code.

Production environments where customers are exposed to bugs and real money is on the line fall into two primary categories, and in general the decision becomes one of balancing cost (amount of CPU and time spent evaluating contract predicates) vs. risk (cost of a failure if a contract is violated unknowingly, or of aborting if a violation makes it past all other testing and brings a system down). Different production systems and different enterprises will want to make these decisions independently.

When performance is not a primary concern, any CCSs that are fast enough to check while also being trusted enough to not be violated by regularly should use the *check_never_continue* semantic. This will catch bugs found by untested configuration or input quickly, but not introduce regular failures for parts of the system that have already been exercised. New CCSs, (for any definition of new relative to the code they checked, as discussed above), however, might stumble upon inputs that were not previously tested with disturbing regularity, and while they remain new they benefit greatly from using the *check_maybe_continue* semantic to "smoke test" the full production environment for contract violations. This semantic should be considered a temporary state when first deploying a new check/set of checks, and once a comfortable enough period of time has passed with no failures they should be moved to the *check_never_continue* semantic along with other checks.

Performance-sensitive applications, where the safety of checking is not worth the cost of checking, face a different set of choices. In these cases, new CCSs likely should be set to the *ignore* semantic until they can be verified as thoroughly as possible externally to the production system. Older, trusted CCSs, however, are a potential place to consider switching to the *assume* semantic to potentially achieve a negative cost for the CCS (removing and optimizing code further based on the additional information being given to the compiler about the values it needs to support).

2 Solutions

[P1332R0] proposed adding a new set of tags on CCSs orthogonal to the *contract-level* to group CCSs based on the above mentioned or other criteria. The complexities of these various concerns and how CSSs might be grouped makes even that proposal not necessarily the best solution. Experimenting with roles on top of the existing proposals seems like the best next step towards deciding what to standardize in the future.

First we will discuss how one might implement a role called `MY_ROLE` on top of the currently existing contract proposals.

2.1 Macros on [N4800]

With the accepted draft standard, roles could be synthesized using a macro that becomes the entire contract attribute, with some relatively limiting constraints. The primary constraint is that *continuation mode* in the draft is a single global flag, so there is no support for having different CCSs in the same TU that continue and that do not continue. Some contortions of the violation handler could be made here, but they will likely not be compatible between different implementations of "roles" from different sources. A second limitation is that there is no way in the current draft to specify a semantic of *ignore* — all unchecked CCSs are assumed. This means any macro framework that wants to use ignore needs to encompass the entire attribute (or at least the predicate) and not just the semantic.

```
#if defined(MY_ROLE_CHECK_MAYBE_CONTINUE)
  #define MY_ROLE_EXPECTS(P) [[ expects : P ]]
  #define MY_ROLE_ENSURES(R, P) [[ ensures R : P ]]
  #define MY_ROLE_ASSERT(P) [[ assert : P ]]
#elif defined(MY_ROLE_CHECK_NEVER_CONTINUE)
  #define MY_ROLE_EXPECTS(P) [[ expects : (P) && "abort" ]]
  #define MY_ROLE_ENSURES(R, P) [[ ensures R : (P) && "abort" ]]
  #define MY_ROLE_ASSERT(P) [[ assert : (P) && "abort" ]]
#elif defined(MY_ROLE_ASSUME)
  #define MY_ROLE_EXPECTS(P) [[ expects axiom : P ]]
  #define MY_ROLE_ENSURES(R, P) [[ ensures axiom R : P ]]
  #define MY_ROLE_ASSERT(P) [[ assert axiom : P ]]
#elif // defined(MY_ROLE_IGNORE) // this role defaults to ignore
  #define MY_ROLE_EXPECTS(P) [[ expects : sizeof( (P)?true:true ) == sizeof(true) ]]
  #define MY_ROLE_ENSURES(R, P) [[ ensures R : sizeof( (P)?true:true ) == sizeof(true) ]]
  #define MY_ROLE_ASSERT(P) [[ assert : sizeof( (P)?true:true ) == sizeof(true) ]]
#endif
```

Then a function would use these macros like this:

```
int foo(int x)
  MY_ROLE_EXPECTS(x > 0)
  MY_ROLE_ENSURES(res, res < 0)
{
  MY_ROLE_ASSERT(x - 1 >= 0)
  return -x;
}
```

This will only behave as expected if the installed violation handler looks at the provided predicate in the *std::contract_violation* it received to see if it ends with the `"abort"` string, and calls `std::abort` if it does. Similarly, the behavior of these macros depends entirely on having the *build level* set to *default* and the *continuation mode* set to *on*.

Most troublesome for this scheme is the possibility that CCSs in a library not using this scheme require a different violation handler behavior to behave correctly, or worse that they provide unsuitable to be turned on and require the build level to be set to *off* to release.

Another disadvantage is that the code generation benefits of the *check_never_continue* semantic — namely that the compiler will know that the predicate is true after the contract check — are highly unlikely to be apparent to the compiler, as it would have to see and unwind the part of the custom violation handler that inspects the predicate for the `"abort"` string.

2.2 Macros on [P1429R0]

[P1429R0] proposed allowing explicit semantics on a contract check. These purposefully allow experimentation with roles without needing to consider the interaction with other build levels/-configuration. Here, a role could be defined as a macro that simply provides a semantic, like this:

```
#if defined(MY_ROLE_CHECK_MAYBE_CONTINUE)
  #define MY_ROLE check_maybe_continue
#elif defined(MY_ROLE_CHECK_NEVER_CONTINUE)
  #define MY_ROLE check_never_continue
#elif defined(MY_ROLE_ASSUME)
  #define MY_ROLE assume
#elif // defined(MY_ROLE_IGNORE)
  #define MY_ROLE ignore
#endif
```

Use of this in a function would then be relatively straightforward:

```
int foo(int x)
  [[ expects MY_ROLE : x > 0 ]]
  [[ expects MY_ROLE res : res < 0 ]]
{
  [[ assert MY_ROLE : x - 1 >= 0 ]]
  return -x;
}
```

The primary advantage of this approach is that the behavior of this role is independent of "regular" use of CCSs, and so the provider of CCSs using these macros does not have to require any particular violation handler behavior or build mode choice.

The disadvantage is that of all macros — name collisions are highly possible, the behavior is not always explicitly clear, and explicit language support for how code might behave with different macros in different TUs is not self-evident.

More importantly, once the macro has expanded the compiler no longer has any information about the intent of the contract check. This means that there is no way to easily distinguish between contract checks that can never be evaluated and those that might be. The downside of that is that when a contract check is *ignored* or *assumed* it will not ODR-use functions it references, which

might lead to cases where a build links successfully with one definition for the macro (either of the unchecked semantics) and fails for a different definition of the macro (either of the checked semantics).

This option, while it reduces the macro to one that simply maps dynamically to a single other name, does remove from the decision about the semantic the knowledge of whether the particular CCS is an `expects`, `ensures` or an `assert`. For users who want to control check enabling based on that particular grouping they either have to either duplicate that information on the line (`[[expects MY_PRE_ROLE : p]]` and `[[ensures MY_POST_ROLE : p]]`) or build the entire attribute from a single macro (`MY_ROLE_EXPECTS(p)`, `MY_ROLE_ENSURES(p)`, etc.).

2.3 Roles

[P1332R0] proposed making roles a first-class value that can be arbitrarily put on CCSs in addition to a level. This would allow CCSs to be written like this:

```
[[ assert my_role : x > 0 ]]
[[ assert audit my_role : x > 0 ]]
[[ assert axiom my_role : x > 0 ]]
```

That proposal does have the downside of not considering whether the limitation of just three levels of contract difficulty will be sufficient for all use cases (though a plethora of roles would be viable to solve that problem). It also does not provide a way to specify what default values might be appropriate for any given role/level combination.

Arbitrary identifiers also have the macro problem of name collisions between libraries. That can be mitigated somewhat by requiring that the identifiers for roles be scoped in some way, perhaps with an identifier like a namespace, leading to checks like this:

```
namespace mylib {
  void foo() {
    [[ assert myroles::my_role : x > 0 ]]
    [[ assert audit myroles::my_role : x > 0 ]]
    [[ assert axiom myroles::my_role : x > 0 ]]
} }
```

This still has the downside of needing to find and determine a configuration for the levels for all roles that might be specified by any contract used by a system.

The namespace could be determined from the namespace of the function the contract is attached to, but in many systems that might lead to an explosion of different roles that need to be configured, or unexpected collisions if configuration can be done by name and independent of namespace.

Finally, often the determination of age of a CCS and of code covered by that CCS is largely based on the library where that code and CCS exist, or the interaction between two libraries (one calling into another). Libraries that do not tag their CCSs with a role would then not be independently configurable to different semantics. One possible solution for that would be to mandate configuration of semantics per-namespace — assigning each CCS the namespace of the function that declares that CCS. Per-module configuration is a different solution that might accomplish the same result,

but again how to specify that and whether it provides the needed granularity without putting undue burden on developers to get useful checking is something that needs to be explored before standardizing something.

Once something like this is adopted into the language, however, configuration at any of the granularities that might be needed will be possible. The compiler will know the role specified, the type of CCS being considered, and any inputs the builder of the application has provided in order to determine the correct semantic.

3 Conclusion

We hope that this discussion helps keep the various concerns about contract configuration in everyone's mind as the discussion of where to go for C++ 20 and beyond with contracts continues.

4 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