Minimizing Contracts

Document #: P1607R0
Date: 2019-03-10
Project: Programming Language C++
Audience: EWG
Reply-to: Joshua Berne <jberne4@bloomberg.net>
Jeff Snyder <jeff-isocpp@caffeinated.me.uk>
Ryan McDougall <mcdougall.ryan@gmail.com>

Abstract

Contracts in C++ 20, how they behave, and how they should be used are currently an open question. The existing proposed *contract levels* of *default*, *audit*, and *axiom* are the result of a great deal of compromise and are not an ideal solution for many of the large institutions hoping to make use of contracts in the language. The currently proposed semantics — choosing between undefined behavior and checking, with a single global flag to control continuation after checking — are equally problematic for use at scale. This paper proposes two solutions to that problem, one involving stripping the contracts feature to a bare minimum to avoid polluting the design space for the future, and one involving adding to that the explicit semantics of [P1429R1] as building blocks for experimentation.

Contents

1 Minimal Solution

I am a great fan of the incremental approach - getting a minimal change in place and then improving it based on feedback. I consider that engineering as opposed to the ideal of getting a change perfect in advance - which I consider naive and at odds with reality.

Bjarne Stroustrup

We propose taking the following parts of the currently proposed solution away from the existing contracts proposal:

- All *contract levels* other than *default* (including any syntax for specifying default).

- Undefined behavior when a contract is not evaluated.
- All values for *build level* other than *off* and *default*
- *continuation mode*, with evaluated contracts never continuing if they fail.

This leaves the following functionality:

- Contract checks can be specified using `[[expects]]`, `[[ensures]]`, and `[[assert]]` (or `[[pre]]`, `[[post]]`, and `[[assert]]` if [P1344R0] is adopted).
- Contracts can be *on*. The predicates will be evaluated and result in an invocation of the *violation handler* if they evaluate to `false`, `std::abort` will be called if the violation handler returns normally.
- The *violation handler* is still establishable in an implementation-defined way.
- Contracts can be *off* and will not be evaluated, but still syntactically checked.

2 Additional Building Blocks

The specific behaviors that are useful for contracts have been inherent in the current form of the contract proposals since [P0542R5]. In 2017, Lisa Lippincott published [P0681R0] with a very thorough analysis of many possible semantics that might be useful for contract checks. [P1429R1] narrowed that focus down to four semantics and proposed allowing explicit use of those semantics by name within contract attributes. We propose adding in four *identifiers with special meaning* and the associated definitions of the semantics for experimenting with different contract behaviors before cementing a higher-level set of functionality into the language on top of those behaviors.

This entails adding the following:

- Four *identifiers with special meaning* that can be placed in a *contract-attribute-specifier* to translate that attribute with a specific semantic — `ignore`, `assume`, `check_never_continue`, and `check_maybe_continue`.
- The definitions of those semantics, wording for which is available in [P1429R1].
- Expand the *build mode* to allow choosing any of the defined semantics for contract attributes with no explicit semantic.

2.1 Example Usage 1: audit

This will allow users who wish to get behavior like the currently proposed *audit* to use a macro like this:

```
#if defined(BUILD_LEVEL_AUDIT) && defined(CONTINUATION_MODE)
  #define AUDIT_SEMANTIC check_maybe_continue
#elif defined(BUILD_LEVEL_AUDIT)
  #define AUDIT_SEMANTIC check_never_continue
```

```
#else
  #define AUDIT_SEMANTIC ignore
#endif
```

Then the following code with a precondition as might be currently specified:

```
T* binsearch(T*begin, T*end, const T&val)
  [[expects audit : is_sorted(begin,end) ]]
```

becomes this:

```
T* binsearch(T*begin, T*end, const T&val)
  [[expects AUDIT_SEMANTIC : is_sorted(begin,end) ]]
```

With control of the behavior based on using `-DBUILD_LEVEL_AUDIT` when compiling.

Similar macros and control macros for any scheme that has been proposed could then easily be built by any enterprise that wishes to use facilities like that, or different, simpler or more complicated facilities.

Once more widespread experience with these systems exists we hope to see a clearer consensus on what to standardize on top of the building blocks proposed here.

2.2 Example Usage 2: assuming

Similarly, users who wish to have a checkable assume could build macro structures like this:

```
#if defined CHECK_SAFER_ASSUME
  #define SAFER_ASSUME(P) [[assert check_never_continue : P]]
#else
  #define SAFER_ASSUME(P) [[assert assume : P]]
#endif
```

This provides a way to target specific points where introduced assumptions help performance, but for testing and diagnosing any issues that might be related to these cases one can switch these assumptions into enforced checks.

3 References

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

[P1429R1] Joshua Berne, John Lakos *Contracts That Work*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1429r1.pdf

[P1344R0] Nathan Myers *Pre/Post vs. Enspects/Exsures*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1344r0.md

[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

[P0681R0] Lisa Lippincott *Precise Semantics for Assertions*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0681r0.pdf