Feedback on the scalability of contract violations handlers in P2900

Document #:	D3191R0
Date:	2024-03-19
Project:	Programming Language C++
Audience:	Evolution, SG 21 (Contracts)

Reply-to:	Louis Dionne <ldionne@apple.com> Yeoul Na <yeoul_na@apple.com> Konstantin Varlamov <varconst@apple.com>

1 Introduction

In the past few years, the LLVM project has pursued various efforts related to improving the security of code written in C and C++. As part of that, we developed `-fbounds-safety` [BOUNDS], a C extension to enforce bounds safety for production C code that executes guaranteed bounds checking for pointer dereferences and terminates the program in case the pointer is used outside of its bounds. We also developed a hardening framework in libc++ [HARDENING], the Clang C++ Standard Library, which allows doing something similar with container iterators as well as validating bounds in e.g. `std::vector::operator[]`.

At their heart, these efforts basically boil down to finding ways to runtime-check existing preconditions on various basic (and less basic) operations involving both language constructs (like dereferencing a raw pointer) and library constructs (like indexing a `std::vector`). While the mechanism for inserting these precondition checks differs from contracts (i.e. they are either compiler-generated or encoded as `_LIBCPP_ASSERT(...)` inside libc++), what happens when a precondition check fails does not differ.

As such, we have experience with implementing and deploying what is basically a contract violation handler mechanism in existing code bases on a very large scale, in millions of lines of production code. Our efforts were deployed in diverse code bases, some of which have relaxed requirements but also many of which are extremely cognizant about code size and performance, as many C++ users are. We believe that this is representative of what C++ users in the industry need.

We are not seeking to standardize these security-related efforts in this paper. However, we recognize that contracts offer the underlying infrastructure for dealing with such pre/post conditions and this paper provides the feedback that we believe needs to be addressed in order for them to be viable at a large scale. Furthermore, we would like to express a lot of excitement towards contracts once they do satisfy these requirements. In particular, we believe that many safety and security related efforts can be built on top of contracts, but only if contracts are designed to be extremely scalable. This, in turn, could have a massive impact on C++'s security posture as demonstrated by our hugely positive experience with hardening so far.

2 Scalability requirements

A contract violation handler mechanism has multiple requirements, which differ based on the user but also the "build mode". These requirements include flexibility in defining a custom contract violation handler, but also a way to handle contract violations with the absolute minimal code size and performance impact. While the first use case is handled very well by the current proposal [**P2900R6?**], the latter is not. We believe targeted changes can be made to P2900 to address these shortcomings.

Specifically, here are the strictest code size requirements we had to satisfy in both libc++ hardening and

`-fbounds-safety`. These requirements basically correspond to what is needed in order to enable contract predicate checks in production in a release build, which we believe should be a primary design goal, if not *the* primary design goal. Of course, these requirements can be relaxed based on the build mode (e.g. debug) and the code base, so in some cases it may be acceptable to have more overhead in return for more flexibility or a better user experience. However, under the strictest requirements:

1. A contract violation should generate no code at all beyond the equivalent of a branch and a `__builtin_trap()`. In particular, this means no call to a weakly-defined function such as a contract-violation handler, no call to a terminating function such as `std::abort`, and no exception-handling code being generated around contract predicates.
2. It should not be necessary to create a `std::contract_violation` object at runtime (either in static storage or otherwise). This is extremely important to avoid massive code bloat. Creating such an object in static storage is akin to generating an RTTI-like structure for each individual contract predicate, which doesn't scale. P2900 technically allows for other implementation strategies, but we do not know how of any other such strategies.
3. To provide a reasonable user experience, the mechanism must still allow for the implementation to retain some information about the contract violation, but that information must not have to live in the executable. In our case, we generate and store it in the debug information.

3 A typical contract violation handler with P2900

First, here is what a typical implementation of the contract violation handling mechanism would look like in a compiler and standard library, based on P2900 along with the recommended practice:

```
// Implicitly declared by the compiler in all translation units
namespace std { inline namespace __1 {
    struct contract_violation;
}}
void handle_contract_violation(std::contract_violation const&);

// Defined in a runtime library (e.g. libc++.dylib)
__attribute__((weak))
void handle_contract_violation(std::contract_violation const&) {
    // log stuff, etc.
}

// What the code using this looks like
template <class T, class Allocator>
reference vector<T, Allocator>::operator[](size_type n) noexcept
  pre [[clang::oops("vector[] index out of bounds")]] (n < size())
{
  return this->__begin_[n];
}
```

Assuming that the current contract semantic is `enforce`, this generates roughly the following code (compiler explorer: https://godbolt.org/z/Pqo7qxKxf):

```
template <class T, class Allocator>
reference vector<T, Allocator>::operator[](size_type n) noexcept {
  {
    static constinit std::contract_violation __info = {
      .comment = "vector[] index out of bounds",
      .location = std::source_location::current(),
      ...
    };
```

```
    bool __violation;
    try {
      __violation = !(n < size());
    } catch (...) {
      ::handle_contract_violation(__info,
                                  std::contract_detection_mode::evaluation_exception);
      std::abort();
    }
    if (__violation) {
      ::handle_contract_violation(__info,
                                  std::contract_detection_mode::predicate_false);
      std::abort();
    }
  }

  return this->__begin_[n];
}
```

This is *a lot* of code and data being generated for a single assertion. In particular, it is interesting to observe that allowing users to redefine the contract violation handler has several immediate consequences. Indeed, we basically need to define the contract violation handler as a weak function, which means that the compiler has to generate a non-inlineable call to it. Also, since the compiler doesn't know the definition of the contract violation handler, it has to provide a reference to a valid `std::contract_violation` object that outlives the predicate, which basically means it needs to emit it inside the executable.

Furthermore, this code has to be generated for each contract predicate. There will be *a lot* of contract predicates. For example, each operation on a standard container or iterator will contain such checks. Each pointer dereference could also potentially generate such checks in a world where we ensure bounds safety at runtime. That goes far beyond just a few user-defined functions, hence the need for this mechanism to be extremely scalable.

4 What we do for hardening and `-fbounds-safety` and why

Here is roughly how we implement these checks in `-fbounds-safety` and libc++ hardening. First, under the strictest code size requirements, we do not allow users to redefine the contract violation handler. The contract violation handler is something that the vendor defines and is then baked in for all users. If the vendor wants to allow for the contract violation handler to be repleaceable, it can provide a contract violation handler that is repleaceable, but it is not required to.

Instead, when a contract predicate is not satisfied, we directly make a call to `__builtin_verbose_trap(message)`. That builtin [TRAP] expands to the smallest instruction on the target platform that will halt the program, just like `__builtin_trap()`. In addition, it creates a special frame in the debug information that is recognized by debuggers and contains the message provided as an argument. That way, when the program crashes due to a contract violation, users can basically load the core dump in a debugger to see where the program crashed and why, but the executable itself never needs to contain information about these possible contract violations. This is significant for code size and this technique is used for libc++ hardening, for the `-fbounds-safety` vendor extension and we also have experience doing this in the Swift Standard Library (where the technique actually originated).

In a nutshell, our approach looks like this:

```
// Defined in libc++
#define _LIBCPP_ASSERT(expression, message)                                    \
  (__builtin_expect(static_cast<bool>(expression), 1)                          \
    ? (void)0                                                                  \
    : _LIBCPP_ASSERTION_HANDLER(some-information-about-failure-location-and-the-message))
```

```
// Defined by vendors, this is the default:
#define _LIBCPP_ASSERTION_HANDLER(message) __builtin_verbose_trap(message)

// What the code using this looks like
template <class T, class Allocator>
reference vector<T, Allocator>::operator[](size_type n) noexcept {
  _LIBCPP_ASSERT(n < size(), "vector[] index out of bounds");
  return this->__begin_[n];
}
```

This generates roughly the following code (compiler explorer: https://godbolt.org/z/q36Gc4W85):

```
template <class T, class Allocator>
reference vector<T, Allocator>::operator[](size_type n) noexcept {
  if (!(n < size()))
    __builtin_verbose_trap("vector[] index out of bounds");
  return this->__begin_[n];
}
```

Not only is this significantly smaller, but it is also more easily optimized and understood by a compiler. For example, optimizations that hoist contract predicate evaluations outside of loops (after inlining a function like `operator[]`) or that remove redundant checks will be necessary for contracts to really scale. We must design contracts for these optimizations to be achievable without requiring a heroic effort.

5 So, what's wrong with P2900?

Not much. P2900 leaves so much as *implementation-defined* behavior that many implementation strategies are viable. In a way, this is great because implementers can decide to do what's right on their platform in niche use cases. On the other hand, it is also important to handle the main use cases out of the box, otherwise even basic usage will require using non-portable extensions (e.g. a non-standard contract semantic).

5.1 Support scalable contract semantics out of the box

We believe that the above way of handling contract violations is basically the most important use case and we think it should be handled out-of-the-box in the Standard, not merely relegated to an implementation-defined contract semantic. This could be done by defining a new contract semantic that checks the predicate, does not invoke the contract violation handler and then terminates in an implementation-defined way if a contract violation occurs.

To frame this suggestion, it is useful to observe that a contract violation handler is basically a fancy logging function. It can do more than that of course, but that's basically the direction that P2900 is pushing us towards, and that's fine. For example, `observe` will run the handler without terminating the program, aka it logs the failure and keeps going. With that in mind, the P2900 basically provides the following contract semantics:

	performs "logging"	**terminates**
`ignore`	no	no
`observe`	yes	no
`enforce`	yes	yes

We believe we simply need to add the missing row in that table, which seems like a really simple and natural change:

	performs "logging"	terminates
`ignore`	no	no
`observe`	yes	no
`??????`	no	yes
`enforce`	yes	yes

5.2 Exceptions in contract assertion predicates

This is another aspect falling out of the scalability requirements. In practice, we expect that exceptions in contract assertion predicates will be rare. However, the current proposal requires the compiler to basically generate a `try-catch` block around contract assertion predicates, which hurts code generation. Possible ways forward:

— Require contract assertion predicates to be `noexcept`.
— Evaluate contract assertion predicates in a `noexcept` context, thus terminating in case an exception is thrown. (Note that `noexcept` doesn't require a `try-catch` to be generated, we can put a flag in the frame and the unwinder terminates).

We understand that this has been discussed in SG 21 before. We believe this needs to be re-evaluated in light of the scalability requirements we put forward.

5.3 Don't force implementations to use `std::abort()`

A previous version of P2900 mentioned that the program would be terminated in an implementation-defined manner after running the contract violation handler. The latest version of the paper mandates that `std::abort()` is used, and we think this is overly prescriptive. Indeed, `std::abort()` raises `SIGABRT` on POSIX platforms, but in some cases you want to actually terminate the program without going through this mechanism. For example, if you suspect the program state might have been compromised by a malicious actor, using a mechanism that can easily be overridden is not desirable.

Beyond that, calling `std::abort()` (which is a library facility typically provided by C Standard Libraries) from a language construct is a bit tricky. Indeed, the compiler can't call `std::abort()` because that would require having the declaration for it. Implementations would likely end up using `__builtin_abort()`, which basically generates a call to an `_abort` symbol to be fulfilled when linking in the C Library. This doesn't always work on freestanding environments, which sometimes define functions like `abort()` as inline functions in their headers.

Overall, we believe we should simply let the implementation decide how to terminate the program. Doing otherwise is too prescriptive and creates too many difficulties for the benefits it offers.

5.4 Replaceability of the contract violation handler

If a user-provided contract violation handler is provided, it has to be provided at link-time since it needs to be used by all contract predicates in the whole program. However, link-time customization is tricky. We often run into issues where shared libraries override `operator new` intending to do so only for their code base, when in reality they end up overriding `operator new` for the whole program. When another shared library (or the application itself) tries to override `operator new` as well, we run into very interesting bugs. Link-time customization is basically an invitation for non-benign ODR violation bugs, and we would like to investigate an alternative for replacing the contract violation handler.

Unfortunately, C++ doesn't provide many tools to solve this kind of problem so this item is not directly actionable.

6 Conclusion

We are really excited with the advancement of contracts. We believe contracts provide the infrastructure to build many impactful features that have serious scalability requirements and we want to ensure that the contract violation handling mechanism is designed from the start with such requirements in mind.

7 References

[BOUNDS] Yeoul Na. `-fbounds-safety` RFC. https://discourse.llvm.org/t/rfc-enforcing-bounds-safety-in-c-fbounds-safety

[HARDENING] Konstantin Varlamov. Libc++ hardening RFC. https://discourse.llvm.org/t/rfc-hardening-in-libc

[TRAP] Konstantin Varlamov. `__builtin_verbose_trap` RFC. https://discourse.llvm.org/t/rfc-adding-builtin-verbose-trap-string-literal