Side Effects of Checked Contracts and Predicate Elision

Document #: P1670R0
Date: 2019-06-16
Project: Programming Language C++
Audience: EWG

Reply-to: Alisdair Meredith <ameredith1@bloomberg.net>
Joshua Berne <jberne4@bloomberg.net>

Contents

Abstract

The current working C++ draft [N4810] declares that most side effects of the predicate of a contract-checking statement (CCS) are considered (language) undefined behavior (UB). This characterization undermines one of the purposes of the contract-checking facility (CCF) by introducing needless pitfalls in its use, some frustrating and some very surprising to many programmers. Instead of punishing users with misbehaving or ill formed programs we propose instead a different solution: requiring the determination of the truth of a predicate when it is invoked while allowing elision of the predicate even if it has side effects.

1 Introduction

The drive to keep contract predicates themselves side-effect free is motivated by the underlying software design principle that valid programs shall have the same observable behavior irrespective of whether runtime contract checking is enabled. The sound engineering intent here is to help prevent programmers from inadvertently writing code that works in only some build modes (when the contract is evaluated and the side effects happen) but fails in others (when code is released to production with contracts no longer checked at runtime, and those side effects turn out to be needed).

In theory, this principle makes perfect sense, and should never be violated. In practice, however, it is inevitable that such side effects in CCS predicates will occur – incidentally (e.g., memory allocation/deallocation), temporarily (e.g., a temporarily inserted print statement, say, for debugging), or inadvertently (e.g., accidentally calling a function out of contract from a predicate). Ideally, we want to do something that helps remedy (or is at least tolerant of) such real-world situations and removes dangerous pitfalls that might make the entire CCF unusable in practice. [1]

In order to take the "bite" out of CCS predicates with side effects, we propose changing the relevant wording to state that – provided there is a way to determine the truthiness of an enabled CCS predicate without executing its code – the evaluation of a predicate can be elided. This would be worded and behave very similarly to how *copy elision* works. The advantages of this formulation are that (1) it leaves all predicates in a state where any side effects they do have cannot be relied upon, (2) allows the optimizer to elide predicates entirely if it can, and (3) avoids making entire classes of seemingly simple contracts dangerously invalid.

Note that this proposal is entirely independent of any others, focusing on only how one addresses side effects in CCSs that are enabled for runtime checking; it should be considered and adopted (or not) independently of any other proposals being put forth.

2 Examples

In this section we show several examples of predicates having side effects which are, in our opinion, benign and making them undefined behavior seems ill-advised.

2.1 Logging

A very common tool when debugging a software problem is to log program state to console output to trace what is happening. Consider the following function `g`, which has had a print statement temporarily added to it for debugging:

[1]The treatment of side effects in CCS is analogous to the impact that template definition checking with concepts would have. It would make typical programming constructs erroneous (e.g., the temporary addition of print statements, calls to unconstrained templates like `std::forward` and `std::move`, or calls to non-template functions like `std::cos`). A compiler that diagnoses UB resulting from side effects in CCS could make programs ill-formed exactly the same way that definition checking would make the program ill-formed when using a declaration not admitted by the template's declared constraints.

```
bool g(int x) {
  std::cout << "g was called!\n";
  return x >= 0;
}
```

Now imagine another function `f` that uses `g` in a precondition:

```
void f(int x)
  [[ expects : g(x) ]];
```

It is important to note that the implementation of a function (such as `g` above) that logs may be very far removed from the client function (such as `f`) performing the precondition check that, at the moment, has a side effect. The function invoked from the predicate (`g` in this case) might even be owned by a different developer. Just by adding a print statement — something with a side effect that does not actually alter the conceptual program state — we have transformed far-removed well-formed code to one that manifestly contains UB.

Perhaps the worst effect of this function now containing UB is also far-reaching — the risk is not limited to the CCS check itself or even what `f` might do. Any program that invokes this `f` will contain UB (for ALL inputs to `f`) and be completely elidable (or worse), e.g., given `f` above, the following program could be built by a conforming compiler with no errors as an empty program that returns anything:

```
int main()
{
  f(0);
  return do_something_important();
}
```

2.2 Contract Violations

As contracts spread throughout a code base it will become more and more likely that predicates of one function invoke other functions that themselves have narrow contracts. Even the simple case of wrapping one set of library functions with another layer (possibly with different names or a slightly different interface, for migration) is likely to introduces scenarios where a CCS predicate itself invokes some other function out of contract. The violation of this other contract, leading to the invocation of the violation handler, will be a side effect, making the entire invocation of the function undefined behavior:

```
bool g(int x)
  [[ expects : x >= 0 ]] ;

void f(int x)
  [[ expects : g(x) ]] ;

int main()
{
  f(-1);
  return 0;
}
```

In the example above, when attempting to build with these contracts checked, it would be ideal to invoke the violation handler and inform the user what has been done wrong (`g` being called out of contract). As currently standardized, the invocation of `f(-1)` has a side effect (invoking the violation handler) and thus this entire program's behavior is undefined.

2.3 Allocation

Next, we'd like to show how simple it is to have an otherwise innocuous side effects without realizing it. Consider a simple `struct` using an `std::map` having the non-transparent comparison function `std::less<std::string>` (the default one):

```
struct x {
  std::map<std::string,std::string> d_map;
  void f(std::string_view key)
    [[ expects : d_map.find(key) != d_map.end() ]];
};
```

The call to `find` in the contract check will require the construction of a temporary `std::string` from the passed in `std::string_view`, (potentially) allocating memory (should the input string not fit into the string's SSO buffer). That allocation is possibly a side effect, depending on whether a custom `operator new` has been linked in, again causing many uses of this function to be categorized as UB.

2.4 constexpr Evaluation

Finally, it's important to note that the impact of making things UB on evaluation in a `constexpr` context is that the compiler needs to identify any execution of such code as ill-formed. This makes all of the above examples unusable in `constexpr` functions.

While adding tracing output is not of use in `constexpr` functions, and contract violations in `constexpr` evaluation should be ill-formed anyway, huge amount of work has and will go on to enable allocations in `constexpr` contexts, and this would prevent functions that try to leverage such allocations from being used in contract predicates.

3 A Better Answer than Undefined Behavior

Designating certain behavior as undefined afford the most flexibility for defining it later, once we have real implementation and use experience with this brand new language-based runtime contract-checking facility. One might say "Undefined behavior is not so bad, the compilers won't do anything wrong because they don't hate us". Though compiler vendors clearly have everyone's best interests at heart, they are also under constant pressure to provide the most optimizations possible, so pressure to leverage all UB will continually grow. Moreover, with the general misunderstanding and fear of undefined behavior in the wider community, it is apparent that any facility which easily lets naive developers stumble into UB is going to be treated warily or outright shunned by many groups. In a world where many companies run sanitizers that flag all UB as bad (and insist that it all be fixed at once), we can imagine that many of those same companies (which should be most interested in using contracts to make there code safer) will instead tend to avoid or disallow the use of C++20 contract checking given how easy it is to inadvertently introduce (language) UB – even with vendors making every effort to keep this particular form of UB benign.

If we had no other, better alternative, we have to settle for making side-effects in CCS predicates UB. There is, however, another solution that (1) establishes the spirit of discouraging meaningful side effects and (2) doesn't introduce additional serious concerns when incidental side effects occasionally manifest. By avoiding making side effects in CCS predicates hard UB, we never open that door.

We propose changing the core wording to state that an implementation may omit the predicate evaluation if it can determine the return value of that CCS, even if the evaluation would have side effects. The practical consequence of this wording is that the compiler can (and in many cases most likely would) just evaluate the predicate (as one would expect), and invoke the violation handler if it returns `false`. In cases where through static analysis the compiler can know the result (say, by chaining the postcondition of one function to prove the precondition of the next) the compiler can instead achieve a conforming result by **not** evaluating the predicate (even though this would have an observable difference since side effects of the predicate will not happen).

4 Other Solutions

One possible (and the simplest) solution for allowing side effects would be to just allow side effects and make no other changes to the existing wording. This would make side effects of checked contracts reliably happen. We believe this would hinder a significant portion of the benefits of using contracts for static analysis (logically chaining postconditions to preconditions to verify at compile time predicate truthiness and remove any need to execute the predicate) while at the same time encouraging users to write and depend on the execution of predicates with side effects. If significant libraries begin to use contract checks to log calls or do other necessary work then many users will be locked into specific build modes to use those libraries and many of the benefits of having build modes will go away.

Another wording that was considered was to state that the evaluation of the predicate is unspecified. Many people this was discussed with reacted as badly to that as they might to undefined behavior, even when the only options allowed for the predicate were the ones that we propose allowing - evaluating the predicate or not.

Formulations allowing for the evaluation of the predicate to be implementation-defined were also considered. We feel it's important to note that the specifics of when a CCS predicate is elided or not be kept flexible and without documented implementation behavior, otherwise the trap of depending on predicate side effects will come back. Compiler vendors should be strongly discouraged from guaranteeing predicate evaluation (or guaranteeing elision) to discourage users from ever depending on predicate side effects.

5 On Exceptions

We suggest leaving the explicit wording about predicates exiting via exception invoking `std::terminate` directly. This is generally to prevent users depending on thrown exceptions from predicate evaluation for flow control.

Note that a predicate that will exit via throwing is also one that cannot be elided - the compiler cannot determine the return value since the branches where the function throws have no return value.

6 Return Value Determination

We have chosen to to present wording that states "When the return value can be determined without evaluating the predicate" to identify when a predicate may be elided. This is with the understanding of the following points about a C++ implementation:

- Expressions with no side effects may be emitted and used at any time as if they were empty statements.
- Any function or variable that is already odr-used may be odr-used again without altering program meaning — i.e., odr-use is idempotent.
- Unobservable expression evaluation may mimic object creation and destruction without becoming side effects — i.e., in the process of evaluating an expression that isn't seen, any variables that might need to live within the lifetime of that hidden expression may do so without violating the rules of the language.

Given these truths, when viewing any predicate that is fully known which either has no side effects or has a subexpression with no side effects which fully determines the return value of the predicate, the compiler can evaluate that side-effect-free subexpression instead of the full predicate in order to satisfy our elision rules.

Consider the following examples of pairs of functions with side effects and equivalent functions that could be used to compute the same return value:

```
bool f()
{
  printf("f called");
  return true;
}
bool f_no_side_effects()
{
  return true;
}

int x;
bool g()
{
  return ++x > 0;
}
bool g_no_side_effects()
{
  int y = x;
  return ++y > 0;
}
```

Two alternative approaches might be considered for wording, and we consider both can be used to communicate what we propose. A normative note outlining the example we propose above could be crafted to make it clear what flexibility is being granted to the compiler when determining the elidability of a predicate. Alternatively, the alternate predicate construction could be explicitly formulated in the normative language, and the wording for elision changed to allow evaluating either the predicate or a suitable equivalent (side-effect free) predicate. We believe any of these solutions would accomplish the same end goal.

7 Unchecked Contracts

With official support for having side effects in checked contract, another question will immediately come up related to whether disabling a contract check definitely disables that side effect. The surprise a developer might experience when an unchecked contract begins logging that its functions are being called - or even without side effects, begins taking very long to execute and determine the truth of contract predicates - should not be discounted.

The current wording for unchecked contracts says "it is unspecified whether the predicate for a contract that is not checked under the current build is evaluated; if the predicate of such a contract would evaluate to false, the behavior is undefined.". When predicates are not allowed to have observable side effects this distinction is irrelevant - there is no way to determine if the compiler chose to evaluate such a predicate or not. With predicates that do have side effects the issue will become very apparent, and the current freedom to evaluate allowed by the draft will become very surprising.

We propose changing this wording to "the predicate for a contract that is not checked under the current build is not evaluated; if the predicate of such a contract would evaluate to false, the behavior is undefined." Note that the same ability the compiler has to determine the value of a predicate to handle elision is what it can use to determine if a predicate is false and treat certain control flows as undefined behavior.

This is orthogonal to other changes in other proposals that might involve making this being undefined behavior conditional on other build modes, but it is tightly related (and the same solution) to the changes that would be necessary to allow `axiom`-level contracts to not ODR-use entities they reference.

8 History

It is important to also note that contracts being undefined behavior came about as a result of CWG questions on what to do about predicates with side effects. The EWG discussion on the subject can be found at `http://wiki.edg.com/bin/view/Wg21jacksonville2018/Contracts-JAX2018`.

During that conversation, after the vote, the suggestion to allow elision was made and the similarity to copy elision was called out by Herb Sutter. Ville Voutilainen, EWG chair, ended the discussion with "If you want to explore this direction, write a paper". This is that paper.

9 Formal Wording

[dcl.attr.contract.syn]/p6 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.~~

After **[dcl.attr.contract.check]/p4** we need to add a paragraph allowing for predicate elision, worded similarly to how **[class.copy.elision]/p1** is worded:

During constant expression evaluation (7.7), only predicates of checked contracts are evaluated. In other contexts, ~~it is unspecified whether~~ the predicate for a contract that is not checked under the current build level is not evaluated; if the predicate of such a contract would evaluate to `false`, the behavior is undefined.

When the return value of the predicate of a checked contract 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*]

[dcl.attr.contract.check]/p5 gets the following small change to allow for a checked contract to not be evaluated (assuming the result of the predicate is determinable):

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~~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.

10 Conclusion

We hope that this paper clarifies the wide range of CCS predicates that should be well defined even if they have side effects, and the changes that would enable that without sacrificing the fundamental goals and principles that the C++ 20 CCF was designed to achieve and uphold.

11 References

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

[P1429R2] Joshua Berne, John Lakos *Contracts That Work*