Contracts on lambdas

Timur Doumler (papers@timur.audio)

Document #: P2890R0
Date: 2023-08-10
Project: Programming Language C++
Audience: SG21

Abstract

This paper proposes to allow contract-checking annotations (preconditions, postconditions, and assertions) on lambda expressions, and explains how name lookup and lambda captures in such annotations should work. The proposed semantics for both are straightforward and consistent with the rest of the language.

1 Introduction

While the work on the Contracts MVP (see [P2388R4]) has so far focused on ordinary functions, there is no a priori reason why we should not allow contract-checking annotations (CCAs) to appear also in lambda expressions. For example, the following code should be well-formed:

```
constexpr bool add_overflows(int a, int b) {
  return (b > 0 && a > INT_MAX - b) || (b < 0 && a < INT_MIN - b);
}

std::vector<int> vec = { /* ... */ };

auto sum = accumulate(vec.begin(), vec.end(), 0, [](int a, int b)
  [[pre: !add_overflows(a, b)]] {
    return a + b;
  });
```

Note that SG21 has not yet settled on a syntax for CCAs. In this paper, we use attribute-like syntax [P2487R0], however the proposal is independent of the choice of syntax and would work in the same way with lambda-based syntax [P2461R1] or condition-centric syntax [P2737R0]. For this reason, we do not specify the exact syntactic position of a CCA on a lambda in this paper; this should be specified in the to-be-adopted Contracts syntax proposal.

2 Name lookup

The most recent revision of the Contracts MVP paper [P2388R4] says the following about CCAs for lambdas:

> These features are deferred due to unresolved issues: [...] a way to express preconditions and postconditions for lambdas: name lookup is already problematic in lambdas in the face of lambda captures. This problem is pursued in [P2036R1], and until it has been solved we see no point in delaying the minimum contract support proposal.

However, since that paper was published, [P2036R3] has been adopted for C++23, which resolved the name lookup issues cited above. We therefore no longer see a problem with allowing all three kinds of CCAs (`pre`, `post`, and `assert`) in lambda expressions. We propose that name lookup for entities inside these CCAs follow the same rules as for lambda trailing return types (see [P2036R3]): name lookup in the CCAs in a lambda first consider that lambda's captures before looking further outward. Consider:

```
int i = 0;
double j = 42.0;
// ...
auto counter = [j=i]() mutable [[pre: j >= 0]] {
  return j++;
};
```

In this code, the `j` in the CCA should refer to the `j` of type `int` introduced by the init-capture, not the `j` of type `double` declared outside. This rule is most consistent with the rest of the language, and least surprising to the user.

3 ODR-use and lambda captures

If we allow CCAs on lambdas, we need to consider how CCAs should interact with the rules for ODR-use and lambda captures.

3.1 Entities in a CCA are always ODR-used

ODR-use of an entity can be observable, both at compile time and at runtime, as it can trigger template instantiations and lambda captures. [P2834R1] proposes the principle that the semantics of a CCA must not affect the proximate compile-time semantics surrounding that annotation. [P2877R0], which we adopted for the Contracts MVP, goes further by removing the notion of build modes and making the semantics of a CCA implementation-defined and in general unknowable at compile time.

From this follows that entities in a CCA must always be ODR-used, even if the semantic of the CCA is *ignore*, because whether an entity is ODR-used cannot depend on the contract semantic. The remainder of this paper therefore assumes that entities in a CCA are always ODR-used. If this principle does not hold, none of the discussion in this paper applies, and we would need to re-think the design.

3.2 ODR-use in a CCA can trigger a lambda capture

ODR-use can trigger lambda captures. It follows that a CCA on a lambda can trigger a lambda capture if it ODR-uses an entity not ODR-used anywhere else. For example, the following CCA will unconditionally trigger a lambda capture:

```
auto f(int i) {
  return sizeof( [=] [[pre: i > 0]] {});
}
```

In this code, `f` will return `sizeof(int)` even if the semantic of the CCA is *ignore*. With the CCA removed, `f` would return `1`.

We believe that this behaviour is most straightforward, most consistent with the rest of the language, and least surprising to the user: it simply falls out of the current rules in C++ for ODR-use and lambda capture.

3.3 Alternative: make triggering a lambda capture from a CCA ill-formed

[P2834R1] proposes to make the above case — a CCA triggering a lambda capture of an entity not otherwise captured — ill-formed. The paper argues that allowing this violates the "zero overhead for ignored predicates" principle. It gives some examples where such a capture can cause an expensive copy of a captured object, and cause a lambda to no longer fit into the small object optimisation of `std::function`. It is therefore possible to construct a program where the mere presence of a CCA, even if its semantic is *ignore*, can cause runtime performance degradation. [P2834R1] goes on to say that this is sufficient justification to make this case ill-formed. Note that, since the contract semantic is in general not observable at compile time, the code above would have to be *unconditionally* ill-formed, regardless of contract semantic,

We do not agree that introducing such a restriction is reasonable. Such cases are unlikely to appear in practice, and if they do, the user gets what they asked for. It is also easily avoidable (don't write the capture). Simply allowing the capture to happen is consistent with how captures work in all other parts of the lambda (the trailing return type and the body). It is also consistent with the rules for `[[assume(expr)]]`, which can cause the same kind of capture — and we already discussed this exact case at length when we standardised `[[assume(expr)]]` (see [P1774R8]). We should let these language features combine naturally, rather than artificially complicating the language rules to micromanage each special case where the user might have got it wrong — we do not do it elsewhere in the Standard, either.

If we were to make this case ill-formed, the user would get a highly non-obvious compiler error and would have to add an extra line ODR-using the entity in question inside the lambda body (for example, casting it to `void`)). Introducing such exceptions and special cases to the basic rules of the language hurts the ergonomics and teachability of C++.

If it turns out that this case is truly relevant, the appropriate solution is to implement a compiler warning for it as a matter of QoI, as is usually done in similar cases. Consider:

```
std::map<int, Widget> map = { /* ... */ };
for (const std::pair<int, Widget>& elem : map)
  // do something with elem
```

In this case, the user got the element type of `std::map` wrong (which is `std::pair<const int, Widget>` rather than `std::pair<int, Widget>`); this generates an unintended implicit conversion, which in turn yields a temporary object that is lifetime-extended by the `const&`. This code compiles and works, but has a silent performance degradation due to the unnecessary conversion and object creation on every iteration of the loop. This is unfortunate; however, we do not add special cases to basic language rules such as range-based `for` loops, implicit conversions, or reference semantics to make such cases ill-formed. Instead, the user gets what they get, and a quality compiler or static analysis tool will issue a warning.

4 Summary

We propose that the following be added to the Contracts MVP (wording to be provided after design approval by SG21):

- Clarify that all entities in a CCA are ODR-used,
- Allow CCAs on lambdas (exact syntactic position to be specified by the to-be-adopted Contracts syntax proposal),
- For CCAs on lambdas, follow the usual C++ rules for ODR-use and lambda captures, like `[[assume]]` does; do not introduce a special exception that entities ODR-used in a CCA but not otherwise should make the program ill-formed if this ODR-use triggers a lambda capture.

References

[P1774R8] Timur Doumler. Portable assumptions. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1774r8.pdf, 2022-06-14.

[P2036R1] Barry Revzin. Change scope of lambda *trailing-return-type*. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2036r1.html, 2021-01-13.

[P2036R3] Barry Revzin. Change scope of lambda *trailing-return-type*. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2036r3.html, 2021-09-14.

[P2388R4] Andrzej Krzemieński and Gašper Ažman. Minimum Contract Support: either *No_eval* or *Eval_and_abort* contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2388r4.html, 2021-11-15.

[P2461R1] Gašper Ažman, Caleb Sunstrum, and Bronek Kozicki. Closure-Based Syntax for Contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2461r1.pdf, 2021-11-15.

[P2487R0] Andrzej Krzemieński. Attribute-like syntax for contract annotations. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2487r0.html, 2021-11-12.

[P2737R0] Andrew Tomazos. Proposal of Condition-centric Contracts Syntax. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2737r0.pdf, 2021-11-15.

[P2834R1] Joshua Berne and John Lakos. Semantic Stability Across Contract-Checking Build Modes. https://wg21.link/p2834r1, 2023-05-15.

[P2877R0] Joshua Berne and Tom Honermann. Contract Build Modes, Semantics, and Implementation Strategies. https://wg21.link/p2877r0, 2023-06-09.