# Contracts on lambdas

Timur Doumler (papers@timur.audio)

Document #: P2890R1
Date: 2023-12-07
Project: Programming Language C++
Audience: SG21

**Abstract**

This paper proposes to allow precondition and postcondition specifiers on lambda expressions. We propose a rule for name lookup inside precondition and postcondition predicates and discuss several alternatives for specifying the semantics of lambda captures triggered by the odr-use of entities inside a contract predicate.

## 1 Introduction

In the Contracts MVP (see [P2900R2]), preconditions and postcondition specifiers are so far only allowed on ordinary functions. However, there is no a priori reason why we should not allow them to appear also on 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);
}

int main() {
  std::vector<int> vec = { /* ... */ };
  auto sum = accumulate(
    vec.begin(), vec.end(), 0,
    [](int a, int b)
      pre (!add_overflows(a, b))  // precondition on lambda
    {
      return a + b;
    });
  // ...
}
```

While not explicitly stated in [P2900R2], we assume that assertions are already allowed on lambdas because they go into the lambda body and are allowed anywhere expressions are allowed.

## 2 Discussion

### 2.1 Name lookup

The most recent discussion of contract annotations on lambdas can be found in [P2388R4]:

> 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 precondition and postcondition specifiers on lambda expressions.

We propose that name lookup for entities inside precondition and postcondition specifiers on lambdas follow the same rules as they do for lambda trailing return types (see [P2036R3]): name lookup first considers the captures of the lambda 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 precondition predicate 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.

## 2.2 Captures

Entities in the predicate of a contract annotation are unconditionally odr-used (see [P2900R2]), regardless of whether the contract annotation is checked or ignored. This must be so because the contract semantic is in general unknown at compile time (see [P2877R0]).

Such odr-use can trigger lambda captures. Therefore, if we let these language features compose naturally and do not introduce any special rules, then a contract annotation on a lambda can trigger a lambda capture if it odr-uses an entity not odr-used anywhere else. This can have observable effects both at runtime and at compile time. Here is an example of a runtime effect (taken from [P2932R2]):

```
std::function<int()> g(const std::vector<S>& v)
{
  int ndx = pickIndexAtRandom(v);
  return [=]()
    pre (0 <= ndx && ndx < v.size()) // needs to capture v
  {
    return ndx; // Obviously we intend this to capture ndx by value.
  };
}
```

Here, the lambda capture triggered by the precondition predicate triggers an expensive copy of the whole vector `v`. Without the precondition specifier being present, the copy would not happen.

Here is an example of a compile-time effect due to a precondition predicate on a lambda triggering a lambda capture:

```
constexpr auto f(int i) {
  return sizeof( [=] pre (i > 0) {} );    // captures i by value
}
```

The same effect would occur with an assertion inside the lambda body:

```
constexpr auto f(int i) {
  return sizeof( [=] { contract_assert (i > 0); } );    // captures i by value
}
```

In both examples above, without the contract annotation, `f` would return `1`, but with the contract annotation added, `f` will return `sizeof(int)`, even if the contract semantic is *ignore*. We can construct a case where merely adding the contract annotation to a lambda triggers the layout of a class to change:

```
struct X {
  char data[f(0)];
};
```

We can even construct an example where the lambda capture triggered by the contract annotation changes which function is selected by overload resolution (taken from [P2932R2]):

```
template <typename T>
std::true_type f(T t);

template <typename T>
std::false_type f(T t)
requires std::is_convertible_v<T, bool(*)()>;

void g() {
  auto x = [=]() { return true; }
  static_assert(!decltype(f(x))::value); // convertible to bool(*)()

  auto y = [=]() pre(x()) { return true; }
  static_assert(decltype(f(x))::value);  // not convertible
}
```

Should we allow such cases?

### 2.2.1 Option 1: Contract predicates can trigger captures

The first option is to not treat contract predicates differently from any other expressions in C++ for the purpose of triggering lambda captures, i.e. to allow the implicit captures to happen in all cases shown above. This behaviour is arguably the most straightforward, most consistent with the rest of the language, and least surprising to the user: it simply falls out of the current rules for odr-use and lambda captures. It is also consistent with assumptions, which behave in the same way:

```
constexpr auto f(int i) {
  return sizeof( [=] { [[assume (i > 0)]]; } );    // captures i by value
}
```

This phenomenon has been extensively discussed when assumptions were standardised for C++23 (see [P1774R8] section 4.4). While there were concerns that `[[assume]]` should not be able to change the layout of a class, ultimately EWG decided it was too much of an edge case unlikely to cause problems in real code to justify complicating the rules of the language to make such a capture ill-formed. The same reasoning can be applied to contract annotations.

In the case where the lambda capture from a contract predicate triggers an expensive copy of the vector, there is a straightforward workaround to avoid the expensive copy: do not use a default-capture, but only capture `v.size()` explicitly, or better still, perform the check outside of the lambda. In the case where the lambda capture from a contract predicate changes the layout of a class, the layout of the class was non-portable in the first place, so user code should not rely on it. Similarly, user code that relies on a lambda *with a default capture* being convertible to a function pointer is inherently broken. In all of those cases, the user wrote bad code and they got what they asked for. It is completely reasonable to say that we do not want to bend over backwards and make the language more complicated in order to make such cases ill-formed.

### 2.2.2 Option 2: Contract predicates do not trigger captures

[P2932R2] proposes that the odr-use of a local entity in a contract predicate should *not* implicitly capture that entity. It argues that allowing the capture would violate the so-called *zero overhead principle*: adding a contract annotation should never lead to a different branch being taken at compile time. Otherwise, this could lead to "heisenbugs" (a program contains a bug, we add a contract annotation to find the bug, but the contract annotation causes the program to take another branch where the bug does not exist) and measurable performance degradations. The paper argues that the existence of cases where the addition of a contract annotation can cause such "heisenbugs" and performance degradations, no matter how obscure these cases are, would be a major disincentive for the adoption of Contracts in C++.

Saying that the addition of contract annotations to a program should never modify the compile-time semantics of the program, such as overload resolution — effectively checking a different program — is a principled and reasonable approach. However, specifying that the capture simply does not happen, as proposed in [P2932R2], leads to the surprising consequence that the wrong variable might end up being used:

```
static int i = 0;

void test() {
  int i = 1;
  auto f = [=] { contract_assert(i > 0); };    // which i is referenced here?
}
```

With the proposal in [P2932R2], the parameter `i` would not be captured, which means that the `i` in the contract predicate would refer to the outer `i` variable which has static storage duration and therefore does not need to be captured in order to be odr-used inside the lambda. We believe that allowing the above behaviour would be very confusing to the user, and that therefore the proposal in [P2932R2] is not viable.

### 2.2.3 Option 3: Triggering a capture from a contract predicate is ill-formed

The better alternative to option 2, avoiding the problem of unintentionally referring to the wrong variable inside the lambda, is to say that if the predicate of a contract annotation triggers a lambda capture of an entity not otherwise captured, the program is unconditionally ill-formed:

```
int i = 0;
auto f(int i) {
  return sizeof( [=] { contract_assert(i > 0); } ); // Error: cannot capture i here
}
```

This satisfies the zero overhead principle for contracts on lambdas and fixes the issue with option 2. We therefore consider option 3 viable. The downsides compared to option 1 are that carving out such an exception for how lambda captures work increases the complexity of the language, and that the resulting compile error might not be obvious for a novice user. Further, if we choose this option, we strongly recommend that we introduce the same rule for assumptions, as a DR against C++23, otherwise option 3 would introduce an unfortunate inconsistency.

### 2.2.4 Option 4: Triggering a capture from a contract predicate emits a warning

Another alternative to address this issue is to not make triggering a lambda capture from a contract predicate ill-formed, but to issue a compiler warning. In C++, there are many other cases where this approach is taken. 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. Such inefficiency 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 these cases ill-formed. Instead, the user gets what they get, a quality compiler or static analysis tool will issue a warning, and several straightforward fixes are available (use the correct type, or just use `auto`), just like in the lambda case we are concerned with here.

We could even go further and make the warning mandated by the standard, rather than just a matter of QoI; we created a precedent for having well-formed programs that nevertheless require a diagnostic by standardising `#warning`, and we might standardise another such case if we adopt the compile-time-*observe* semantic from [P2894R1].

However, triggering implicit captures from contract predicates is arguably somewhat different from the `std::map` example. As we saw above, triggering a capture from a contract predicate is *always* suboptimal and questionable code, not just in certain cases. It therefore seems that, if we do not want to allow the user to just "get what they get" (option 1) in those cases, then making them ill-formed (option 3) is a better approach than issuing a warning (option 4). Option 3 is a more effective and more teachable deterrent to writing such code and we do not have to worry about false positives.

#### 2.2.5 Option 5: odr-using an entity in a contract predicate that is not otherwise odr-used is ill-formed

Triggering lambda captures is actually not the only way in which the odr-use of an entity in a contract predicate can violate the zero overhead principle from [P2932R2]. Another way is through triggering a template instantiation, which in turn can change observable state via friend injection. This case might be negligible because it is an obscure wart of the C++ language that we would like to make ill-formed (see [CWG2118]). But there is a third and much more plausible case where odr-use of an entity in a contract annotation can violate the zero overhead principle: a template instantiation can trigger dynamic initialisation of a static class member, which in turn can have arbitrary effects on the observable behaviour of the program:

```
struct X {
  X() { std::cout << "This is a bug!\n"; }
  operator bool() const;
};

template <typename T>
struct Y {
  static const X x;
};

template <typename T>
const X Y<T>::x = X{};

void f()
  pre(Y<int>::x);   // the addition of this contract annotation triggers the bug!

int main() {
  f();
}
```

Therefore, making it ill-formed for a contract predicate to trigger a lambda capture (option 3) would not actually be a comprehensive fix that ensures odr-use of entities inside a contract predicate

never violate the zero overhead principle. Such a comprehensive fix would be to make any program ill-formed where a contract predicate odr-uses an entity not otherwise odr-used (and not just if that odr-use triggers an implicit lambda capture). This is our option 5.

We believe that this option 5 is impractical, as it would prohibit the user from e.g. using STL algorithms like `std::for_each` in a contract annotation, unless the exact same instantiation of `std::for_each` is already used somewhere else in the program. Further, it is not obvious how this option could be implemented as such odr-use might occur in a different translation unit.

Note that triggering a template instantiation, unlike triggering a lambda capture, while arguably a violation of the zero overhead principle, does not seem to pass the litmus test in Appendix A of [P2932R2] as it cannot affect overload resolution. It therefore seems like a less serious violation of the zero overhead principle than the lambda capture case.

#### 2.2.6 Option 6: Do not allow contracts on lambdas with default captures

Since the issue discussed here only occurs with lambda expressions that have default captures, we could also say that contract annotations should not be allowed on such lambdas. Note that this would affect not only `pre` and `post`, for which the possibility to add them to lambdas is being proposed in this paper, but also any `contract_assert` in the body of the lambda, which the Contracts MVP permits today.

We believe that this solution is too drastic as it would take away too much value for the user, therefore we do not consider this solution viable.

#### 2.2.7 Option 7: Do not allow contracts on lambdas

As the most radical solution to the problem, we could say that contract annotations are not allowed on any lambdas (tcodepre and `post`) or inside the body of any lambdas (`contract_assert`). We do not consider this solution viable for the same reason as option 6.

## 3 Summary

We propose to allow adding precondition and postcondition specifiers to lambda expressions in the Contracts MVP [P2900R2].

We propose that name lookup for entities inside precondition and postcondition predicates on lambdas follow the same rules as they do for lambda trailing return types: name lookup first considers the captures of the lambda before looking further outward.

Entities in a contract predicate are odr-used, even if the contract annotation has the *ignore* semantic. This odr-use inside a lambda can trigger an implicit lambda capture. It follows therefore that adding a contract annotation to a program can trigger a lambda capture that would otherwise not happen, which in turn can change the compile-time semantics of the program. This violates a desirable property called the *zero overhead principle* (see [P2932R2]). To address this issue, we have five options:

1. Contract predicates can trigger captures (the default if we do not add any special rule).
2. Contract predicates do not trigger captures (proposed by [P2932R2]).
3. Triggering a capture from a contract predicate is ill-formed.
4. Triggering a capture from a contract predicate emits a warning.
5. odr-using an entity in a contract predicate that is not otherwise odr-used is ill-formed.

6. Do not allow any contract annotations on lambdas with default captures

7. Do not allow any contract annotations on any lambdas

For the reasons discussed in this paper, we believe that the only viable options are 1 and 3. Option 1 is the correct choice if we prefer to not add additional complexity to C++, and accept that one can construct cases where the zero overhead principle from [P2932R2] might be violated by captures from entities inside contract predicates (with straightforward fixes available for such cases). Option 3 is the correct choice if there should be *no possible case* where contract predicates would capture entities in a way that violates the zero overhead principle from [P2932R2]. Note however that there are other circumstances, unrelated to lambda captures, where odr-use inside a contract predicate can still violate the zero overhead principle, in a way not addressed by option 3 or any other option in this paper (although arguably those are less serious violations of the principle).

If we choose option 3, we strongly recommend that we introduce the same rule for assumptions, as a DR against C++23, to be consistent.

## References


[CWG2118] Richard Smith. Core Issue 2118: Stateful metaprogramming via friend injection. https://cplusplus.github.io/CWG/issues/2118.html, 2022-06-27.

[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.

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

[P2894R1] Timur Doumler. Constant evaluation of Contracts. https://wg21.link/p2894r1, 2023-12-04.

[P2900R2] Joshua Berne, Timur Doumler, and Andrzej Krzemieński. Contracts for C++. https://wg21.link/p2900r2, 2023-11-11.

[P2932R2] Joshua Berne. A Principled Approach to Open Design Questions for Contracts. https://wg21.link/p2932r2, 2023-11-14.