Lambdas in unevaluated contexts

Document #:	P0315R0
Date:	2016-05-24
Project:	Programming Language C++ Evolution Group
Reply-to:	Louis Dionne <ldionne.2@gmail.com>

1 Introduction

Lambdas are a very powerful language feature, especially when it comes to using higher-order algorithms with custom predicates or expressing small, disposable pieces of code. Yet, they suffer from one important limitation which cripples their usefulness for creative use cases; they can't appear in unevaluated contexts. This draconian restriction was originally designed to prevent lambdas from appearing in signatures, which would have opened a can of worm for mangling because lambdas are required to have unique types. However, that restriction was neither necessary nor sufficient to implement that intent, a situation that has been resolved when core issue 1607 was addressed in C++14. Furthermore, as the core issue states, the restriction is now stale, since the core issue was addressed:

> If any of these approaches were adopted, the rationale for disallowing lambda-expressions in unevaluated operands would be removed, so it might make sense to remove the restriction at the same time.

This paper proposes repairing that oversight and lifting the restriction that lambdas can't appear in unevaluated contexts.

2 Motivation

The original use case that motivated this article is related to making algorithms on heterogeneous containers more useful. For a bit of background, it is possible to implement `std`-like algorithms that operate on `std::tuples` instead of usual, runtime sequences. For example, it is possible to write an algorithm akin to `std::sort`, but which works on a `std::tuple` instead of a runtime sequence:

```
// Returns a new tuple whose elements are sorted according to the given
// binary predicate, which must return a boolean `std::integral_constant`.
template <typename ...T, typename Predicate>
auto sort(std::tuple<T...> const& tuple, Predicate const& pred);
```

The algorithm can then be used as follows:

```
auto tuple = std::make_tuple(std::array<int, 5>{}, 1, '2', 3.3);
auto sorted = sort(tuple, [](auto const& a, auto const& b) {
  return std::integral_constant<bool, sizeof(a) < sizeof(b)>{};
});
// sorted is now a std::tuple<char, int, double, std::array<int, 5>>
```

While this is a simplified example, it is also possible to define other algorithms like `for_each`, `transform`, `accumulate`, `find_if` and many more. This is exploited extensively in the [Boost.Hana] library, which provides high-level algorithms and data structures to make metaprogramming more structured.

Where the current proposal meets with the above use case is when one needs the type resulting from an algorithm exposed above. For example, to get the type of the above tuple without actually creating the tuple, one would like to simply write

```
using sorted = decltype(sort(tuple, [](auto const& a, auto const& b) {
  return std::integral_constant<bool, sizeof(a) < sizeof(b)>{};
}));
```

Unfortunately, with the current restriction on lambdas, this is impossible. Instead, one must create a variable holding the lambda, and then pass this variable to the algorithm:

```
auto predicate = [](auto const& a, auto const& b) {
  return std::integral_constant<bool, sizeof(a) < sizeof(b)>{};
};
using sorted = decltype(sort(tuple, predicate));
```

Unfortunately, this is both clumsy and not always possible since some contexts do not allow defining local variables (for example inside a class declaration). Hence, the restriction severly reduces the usefulness of lambdas in these algorithms. Also note that the issue presented above does not only arise in the context of manipulating heterogeneous containers. Indeed, one could just as well try to write the following, only to be puzzled by a compiler error:

```
std::vector<int> v{1,2,3,4};
using Iterator = decltype(std::find_if(begin(v), end(v), [](int i) {
  return i % 2 == 0;
}));
```

While this is a valid use case, it is expected that using `decltype` on such a complex expression is less frequent outside the realm of heterogeneous computations.

Finally, another motivation for this paper is that the restriction is now obsolete, yet it needlessly prevents lambdas from being used in creative ways, some of which are certainly unknown to the author of this paper. Quoting an exchange between Richard Smith and Roland Bock on the std-proposal mailing list:

```
> > > > If we remove the (now-redundant) restriction on lambdas in unevaluated
> > > > operands, you could write: [...]
> > >
> > > Are there plans/proposals to do that? I stumbled over this restriction
> > > quite a few times.
> >
> > I don't think it's on the core issues list. We discussed it while working
> > on core issue 1607, but I think we thought it would be best handled
> > separately.
>
> Can you give me a pointer as to why the restriction is redundant now?

See core issue 1607. The purpose of the restriction was to avoid lambda-
expressions appearing in a signature (so that implementations don't need to
implement declaration / statement SFINAE and don't need to mangle them), but
it was neither necessary nor sufficient for that. The new rules in core issue
1607 don't need this restriction to implement that intent.
```

Hence, we feel like it would now be reasonable for the restriction to be lifted.

3 Proposed Wording

The proposed wording is probably incomplete, for the author lacks sufficiently deep knowledge of the standard to be sure that no other sections are impacted. However, preliminary survey suggests that only the following change would be required (based on the working paper [N4296]):

> **In [expr.prim.lambda] 5.1.2/2:** Remove *unevaluated operand* from the list of things a lambda-expression may not appear in.

4 Implementation Experience

This proposal was implemented in Clang. The required change is commenting a single line which creates a diagnostic if a lambda-expression is found inside an unevaluated context.

5 Acknowledgements

Roland Bock and Matt Calabrese for discussing use cases for lambdas in unevaluated contexts on the *std-proposal* and *Boost.Devel* mailing lists. Richard Smith for letting me know that the restriction was now redundant.

6 References

[Boost.Hana] Louis Dionne, *Boost.Hana, A modern metaprogramming library*
`https://github.com/boostorg/hana`

[N4296] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf