Lambdas in unevaluated contexts

Document #:	D0315R1
Date:	2016-07-07
Project:	Programming Language C++ Evolution Group

Reply-to:	Louis Dionne <ldionne.2@gmail.com>

1 Introduction

- R0 – Initial draft
- R1 – Changed the wording to work around the resolution of DR1607, which conflicted with the initial wording.

2 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 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, the restriction is much stronger than it needs to be, and it is indeed possible to achieve the same effect without it, as evidenced by this paper.

3 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::tuple`s 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 much stronger than it needs to be, and it prevents lambdas from being used in creative ways, some of which are certainly unknown to the author of this paper.

4 Proposed Wording

The wording is based on the working paper [N4606].

- Modify [**decl.typedef**] **7.1.3/9** as follows:

> If the typedef declaration defines an unnamed class (or enum), the first typedef-name declared by the declaration to be that class type (or enum type) is used to denote the class type (or enum type) for linkage purposes only (3.5). However, a closure type is never given a name for linkage purposes. [*Example:*
>
> ```
> typedef struct { } *ps, S; // S is the class name for linkage purposes
> typedef decltype([]{}) C; // the closure type has no name for linkage purposes
> ```
>
> – *end example*]

The intent is to clarify that closure types are never given a name for linkage purposes, thus preventing their use inside names with external linkage.

- At the end of [**temp.alias**] **14.5.7**, add the following paragraph:

> The type of a lambda expression appearing in an alias template declaration is unique across instantiations of that alias template, even when the lambda expression is not dependent. [*Example:*
>
> ```
> template <typename T>
> using A = decltype([] {});
> // A<int> and A<char> refer to different closure types
> ```
>
> – *end example*]

- After [**temp.deduct**] **14.8.2/8**, add the following note:

> [*Note:* The body of a lambda expression appearing in a function type or a template parameter is not considered part of the immediate context for the purposes of template argument deduction. The intent is to avoid requiring implementations to deal with substitution failure involving arbitrary statements. [*Example:*
>
> ```
> template <typename T>
> auto f(T) -> decltype([]() { T::invalid; } ());
> void f(...);
> f(0); // the invalid expression is not part of the immediate context, hard error
>
> template <typename T, std::size_t = sizeof([]() { T::invalid; })>
> void g(T);
> void g(...);
> g(0); // the invalid expression is not part of the immediate context, hard error
>
> template <typename T>
> auto h(T) -> decltype([]() -> std::void_t<typename T::invalid> { });
> void h(...);
> h(0); // deduction fails on #1, calls #2
>
> template <typename T>
> ```

```
auto i(T) -> decltype([](){ } (t));
void i(...);
i(0); // deduction fails on #1, calls #2
```

– *end example*] – *end note*]

Note that the term "immediate context" is not defined formally in the standard, which is the subject of CWG1844.

- In **[expr.prim.lambda] 5.1.2/2** modify the paragraph as follows:

> The evaluation of a lambda-expression results in a prvalue temporary (12.2). This temporary is called the closure object. A lambda-expression shall not appear in an unevaluated operand (Clause 5), in a template- argument, in an alias-declaration, in a typedef declaration, or in the declaration of a function or function template outside its function body and default arguments. [*Note:* The intention is to prevent lambdas from appearing in a signature. – *end note*] [*Note:* A closure object behaves like a function object (20.9). – *end note*]

5 Implementation Experience

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

6 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 we could do without the restriction. David Vandervoorde and Hubert Tong for providing extensive guidance in the second iteration of the wording.

7 References

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

[N4606] Richard Smith, Working Draft, Standard for Programming Language C++
https://github.com/cplusplus/draft/blob/master/papers/n4606.pdf