Lambdas in unevaluated contexts

Document #:	D0315R2
Date:	2017-04-08
Project:	Programming Language C++
Audience:	Core Working Group

Reply-to:	Louis Dionne <ldionne.2@gmail.com>
	Hubert Tong <hubert.reinterpretcast@gmail.com>

1 Revision history

- R0 – Initial draft
- R1 – Changed the wording to work around the resolution of [DR1607], which conflicted with the initial wording. Also address the potential additional concerns raised by this wording change.
- R2 – The following changes were made per feedback from EWG and CWG in Issaquah and CWG in Kona:
  - a lambda expression is not part of the immediate context
  - add a discussion about lambdas as non-type template arguments
  - clarify the difference between the types of lambda expressions in alias templates
  - clarify ODR-equivalence of lambda expressions declared in different TUs, and the meaning for function template declarations
  - clarify the situation for redeclarations of functions with lambda-expressions
  - rebase on top of the C++17 DIS
  - extract the wording into its own section (editorial)
  - TODO: add discussion about implicit captures in unevaluated contexts

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 Discussion

The core language changes introduced in this paper are a bit tricky. The reason is that we remove many restrictions on lambda expressions, yet we still want to keep closure types out of the signature of external functions, which would be a nightmare for implementations. This discussion goes over all known possible points of contingency to clarify them.

1. (wording) With the removal of the restrictions on lambdas in unevaluated contexts, a concern is that lambda-expressions might then be able to appear in the signature of functions with external linkage, which would require implementations to generate a name for the associated closure type. However, since we wouldn't be able to attach to another ABI entity in some cases (such as lambda-expressions appearing at global scope), that would mean coming up with a mangling scheme that identifies the closure type from nothing but its own form. This, in turn, would require encoding its complete definition, which is burdensome for implementations and motivated the original restrictions on lambda-expressions.

   Fortunately, this specific problem can't arise in the case of non-template functions, even with the above removal of constraints on lambda-expressions. Indeed, according to **[basic.link] 6.5/8**, closure types have no linkage, and therefore they cannot appear in the signature of a function with external linkage (a function is a compound type):

   > *[...]* A type is said to have linkage if and only if:
   > *[...]*
   > - it is a compound type (6.9.2) other than a class or enumeration, compounded exclusively from types that have linkage; or
   > *[...]*

   However, to make it clear that closure types are never given a name for linkage purposes, we propose modifying **[dcl.typedef]**.

2. (wording) Another similar problem is that of lambda-expressions appearing in the signature of function templates. There are two ways this could happen. First, a lambda expression could appear not by itself, but indirectly, by being part of an expression which references a template parameter. Indeed, per **[temp.over.link] 17.5.6.1/4**:

> When an expression that references a template parameter is used in the function parameter list or the return type in the declaration of a function template, the expression that references the template parameter is part of the signature of the function template.

Thus, a function template declaration such as the following will require the implementation to make the lambda-expression part of the signature, which is specifically what we would like to avoid:

```
template <int N>
void f(const char (*s)[([]{}, N)]) { }
```

The other situation we would like to avoid is for implementations to have to figure out that the two following expressions are equivalent, either for linkage purposes (in different translation units) or for redeclaration purposes (in the same translation unit):

```
template <int N> void f(const char (*s)[([]{ return N; })()]) { }
template <int N> void f(const char (*s)[([]{ return N; })()]) { }
```

This case is slightly different from the first one, since the template parameter appears in the body of the lambda-expression, which is not considered part of the full-expression in the function parameter list. To make sure these cases do not happen, we propose amending **[temp.over.link]**.

3. Another possible concern is the appearance of lambda-expressions in contexts that are constrained by the ODR. For example:

```
// a.h:
template <typename T>
int counter() {
  static int cnt = 0;
  return cnt++;
}

inline int f() {
  return counter<decltype([] {})>();
}

// translation unit 1:
#include "a.h"
int foo() { return f(); }

// translation unit 2:
#include "a.h"
int bar() { return f(); }
```

Given such code, a question might be whether `foo` and `bar` modify the same `cnt` variable, since `f` is defined in a header and it calls `counter` with a closure type that is supposed to be unique. However, since `f` is `inline`, the resulting program is as-if there was a single definition

of it, and so both functions end up modifying the same `cnt` variable. This turns out not to be a problem for implementations, because they must already handle such cases where there is an ODR context to attach the closure type to. Thus, no wording change is required.

4. (wording) A concern with allowing lambda-expressions in *declarations* is that of dealing with entities that can be redeclared. It is already the case that no two lambda-expressions share the same closure type within a single translation unit:

```
static decltype([] { }) f();
static decltype([] { }) f(); // invalid; return type mismatch

static decltype([] { }) g();
static decltype(g()) g(); // okay

static void h(decltype([] { }) *) { }
static void h(decltype([] { }) *) { }
h(nullptr); // ambiguous

using A = decltype([] { });
static void i(A *);
static void i(A *) { }
i(nullptr); // okay
```

By further clarifying that the lambda-expressions in alias template specializations are unique to each specialization even if non-dependent, we conclude the following:

```
template <typename T>
using B = decltype([] { });
static void j(B<char16_t> *) { }
static void j(B<char32_t> *) { }
j(nullptr); // ambiguous
```

To make the above interpretation of the standard more obvious, we propose modifying **[temp.alias]**.

Furthermore, some questions were raised on the Core reflector regarding redeclarations like this:

```
template <int N> static void k(decltype([]{ return 0; }()));
template <int N> static void k(decltype([]{ return 0; }())); // okay
template <int N> static void k(int);                         // okay
```

These should be valid redeclarations, since the lambda expressions are evaluated, and they neither contain a template parameter in their body nor are part of a full expression that contains one. Hence, the lambda-expression does not need to appear in the signature of the function, and the behavior is equivalent to this, without requiring any special wording:

```
struct lambda { auto operator()() const { return 0; } };
template <int N> static void k(decltype(lambda{}()));
```

```
template <int N> static void k(decltype(lambda{}())); // okay today
template <int N> static void k(int);                  // okay today
```

5. (wording) A concern with allowing lambda-expressions *outside the body of the declaration of function templates* is the need to evaluate the validity of potentially complex expressions as part of template argument deduction. Indeed, without clarifying the wording, it is unclear whether implementations would be expected to support *SFINAE* based on the validity of the body of a lambda-expression found in the declaration of a function template. Since this could be unwieldy for implementations, we choose not to require this in the current paper. Thus, if a lambda-expression appears inside the declaration of a function template and any part of it is ill-formed, then the program is ill-formed. To reflect this, we propose adding a note at the end of **[temp.deduct]**.

6. One usability question that has been raised with this paper is related to the usage of lambda expressions as non-type template arguments.

```
// foo.h
template <auto> struct foo { };
foo<[]() {}> x;

// translation unit 1:
#include "foo.h"

// translation unit 2:
#include "foo.h"
```

   With the current wording that would be an ODR violation. Is this something we want to do something about?

5 Implementation Experience

This proposal was naively 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.

6 Proposed Wording

The wording is based on the C++17 DIS [N4659]:

1. In **[expr.prim.lambda] 8.1.5/2**:

> 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 8), 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 (23.14). – end note]

2. (discussion) In **[dcl.typedef] 10.1.3/9**:

> 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 (6.5). 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*]

3. (discussion) In **[temp.over.link] 17.5.6.1/5**:

> Two expressions involving template parameters are considered *equivalent* if two function definitions containing the expressions would satisfy the one-definition rule (6.2), except that the tokens used to name the template parameters may differ as long as a token used to name a template parameter in one expression is replaced by another token that names the same template parameter in the other expression. Unless both expressions occur such that they are corresponding counterparts of each other within definitions of the same enclosing entity in separate translation units ([**basic.def.odr**]), two lambda-expressions appearing in full expressions involving template parameters are never considered equivalent, and two lambda-expressions involving within their bodies template parameters are never considered equivalent. [*Note:* The intent is to avoid lambda-expressions appearing in the signature of a function template with external linkage. – *end note*]

Also, add the following example after **[temp.over.link] 17.5.6.1/5**:

```
// ill-formed, no diagnostic required:  the two expressions are
// functionally equivalent but not equivalent
template <int N> void foo(const char (*s)[([]{}, N)]); // TU 1
template <int N> void foo(const char (*s)[([]{}, N)]); // TU 2

// ill-formed, no diagnostic required:  the two lambda-expressions are
// functionally equivalent but not equivalent
template <int N> void bar(const char (*s)[([]() { return N; })()]); // TU 1
template <int N> void bar(const char (*s)[([]() { return N; })()]); // TU 2
```

4. (discussion) Add the following paragraph at the end of **[temp.alias] 17.5.7**:

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

– end example]

5. (discussion) Add the following after **[temp.deduct] 17.8.2/8** (note that the term *immediate context* is not defined formally in the standard, which is the subject of [CWG1844]):

> 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. [*Note:* The intent is to avoid requiring implementations to deal with substitution failure involving arbitrary statements. [*Example:*
>
> ```
> template <class T>
> auto f(T) -> decltype([]() { T::invalid; } ());
> void f(...);
> f(0); // error:  invalid expression not part of the immediate context
>
> template <class T, std::size_t = sizeof([]() { T::invalid; })>
> void g(T);
> void g(...);
> g(0); // error:  invalid expression not part of the immediate context
>
> template <class T>
> auto h(T) -> decltype([x = T::invalid]() { });
> void h(...);
> h(0); // error:  invalid expression not part of the immediate context
>
> template <class T>
> auto i(T) -> decltype([]() -> typename T::invalid { });
> void i(...);
> i(0); // error:  invalid expression not part of the immediate context
>
> template <class T>
> auto j(T t) -> decltype([](auto x) -> decltype(x.invalid) { } (t));
> void j(...);
> j(0); // deduction fails on #1, calls #2
> ```
>
> *– end example*] *– end note*]

7 Acknowledgements

Thanks to Roland Bock and Matt Calabrese for discussing use cases for lambdas in unevaluated contexts on the *std-proposal* and *Boost.Devel* mailing lists. Thanks to Richard Smith for letting me know that we could do without the restriction. Thanks to Hubert Tong and David Vandevoorde for providing extensive guidance for the wording.

8 References

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

[N4659] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

[DR1607] Daniel Krügler, *Lambdas in template parameters*
http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#1607

[CWG1844] Richard Smith, *Defining "immediate context"*
http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1844