Familiar template syntax for generic lambdas

Document #:	D0428R0
Date:	2016-08-21
Project:	Programming Language C++
Audience:	Evolution Working Group

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

1 Introduction

C++14 added the ability to define generic lambdas, i.e. lambdas where the `operator()` of the generated *closure-type* is a template. This addition was initially proposed in [N3418], which included many different features for generic lambdas, including the functionality proposed by this paper. However, N3418 was not accepted as-is and its successor, [N3559], was accepted instead. N3559 settled on the `auto`-based syntax that we know in C++14 for defining generic lambdas, leaving the usual template syntax out for lack of clear use cases (according to an author of N3559):

```
[](auto x) { /* ... */ }
```

Unfortunately, this syntax makes it difficult to interact with the type of the parameter(s) and lacks flexibility that is sometimes required, as outlined in the **Motivation** section. Hence, this paper proposes adding the ability to use the familiar template syntax when defining lambda expressions:

```
[]<class T>(T x) { /* ... */ }
[]<class T>(T* p) { /* ... */ }
[]<class T, int N>(T (&a)[N]) { /* ... */ }
```

2 Motivation

There are a few key reasons why the current syntax for defining generic lambdas is deemed insufficient by the author. The gist of it is that some things that can be done easily with normal function templates require significant hoop jumping to be done with generic lambdas, or can't be done at all. The author thinks that lambdas are valuable enough that C++ should support them just as well as normal function templates. The following details such areas where lambdas are lacking in their current form:

1. It is often useful to retrieve the type of the parameter of a generic lambda, e.g. for accessing a static member function or an alias nested inside it. However, retrieving such a type requires using `decltype`, which includes its reference and cv qualifiers. This can often lead to unexpected results:

```
auto f = [](auto const& x) {
  using T = decltype(x);
  T copy = x;                                // Compiles, but wrong semantics!
  T::static_function();                      // Does not compile!
  using Iterator = typename T::iterator;     // Does not compile!
};
```

To work around this unfortunate situation, one must introduce some amount of verbosity:

```
auto f = [](auto const& x) {
  using T = std::remove_cv_t<std::remove_reference_t<decltype(x)>>;
  T copy = x;
  T::static_function();
  using Iterator = typename T::iterator;
};
```

Furthermore, this problem compounds when trying to make a parameter type dependent on a previous parameter type, because aliases can't be introduced in that context to reduce verbosity:

```
auto advance = [](auto& it,
                  typename std::remove_reference_t<decltype(it)>::difference_type n) {
  // ...
};
```

Instead, it would be much nicer to simply write

```
auto f = []<class T>(T const& x) {
  T copy = x;
  T::static_function();
  using Iterator = typename T::iterator;
};

auto advance = []<class It>(It& it, typename It::difference_type n) {
  // ...
};
```

2. Perfect forwarding in generic lambdas is more verbose than it needs to be, and the syntax for it is different from what's usually done in normal function templates. While this is technically a direct corollary of the previous point, the author thinks this is sufficiently annoying to be worth mentioning separately. The problem is that since the only way to get an argument's type in a lambda is to use decltype, we must resort to the following syntax for perfect forwarding:

   ```
   auto f = [](auto&& ...args) {
     return foo(std::forward<decltype(args)>(args)...);
   };
   ```

   Exactly why this works is explained in a blog post written by Scott Meyers [Meyers], but the very fact that Meyers had to write a blog post about it is telling. Indeed, the interaction between template argument deduction and reference collapsing rules is already sufficiently

complicated that many C++ users would benefit from the cognitive load reduction allowed by a single perfect forwarding syntax for both lambdas and normal functions:

```
auto f = []<class ...T>(T&& ...args) {
  return foo(std::forward<T>(args)...);
};
```

3. The limited form of "pattern matching" on template argument allowed by C++ in function templates is very useful, and it would be equally useful to allow it in lambda expressions. For example, writing a lambda that accepts a `std::vector` containing elements of any type (but not another container) is not possible with the current syntax for generic lambdas. Instead, one must write a catch-all generic lambda that accepts any type, and then assume that it is of the proper type, or check that it is not through other means:

```
template <class T> struct is_std_vector                 : std::false_type { };
template <class T> struct is_std_vector<std::vector<T>> : std::true_type { };

auto f = [](auto vector) {
  static_assert(is_std_vector<decltype(vector)>::value, "");
};
```

In addition to being verbose, calling the lambda with a type that is not a `std::vector` will result in a hard error inside the body of the lambda, not a template argument deduction failure. This does not play nicely with other parts of the language such as SFINAE-based detection, and it is obviously not as clear as the equivalent function template.

Another instance where "pattern matching" would be useful is to deconstruct the type of arguments that are template specializations. For example, imagine that we want to get the type of elements stored in the vector in the previous example. Right now, we'd have to write this:

```
auto f = [](auto vector) {
  using T = typename decltype(vector)::value_type;
  // ...
};
```

This is cumbersome syntax-wise, and it requires the type to provide a nested alias that does just the right thing. This is not a problem for `std::vector`, but most types don't provide such aliases (and in many cases it wouldn't make sense for them to). Hence, right now, types that do not provide nested aliases or accompanying metafunctions can simply not be deconstructed in lambdas. Instead, it would be much simpler and more flexible to write

```
auto f = []<class T>(std::vector<T> vector) {
  // ...
};
```

3 Proposed Wording

TODO

4 Acknowledgements

TODO

5 References

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

[Meyers] Scott Meyers, C++14 Lambdas and Perfect Forwarding
http://scottmeyers.blogspot.com.tr/2013/05/c14-lambdas-and-perfect-forwarding.html

[N3418] Faisal Vali, Herb Sutter, Dave Abrahams, Proposal for Generic (Polymorphic) Lambda Expressions
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3418.pdf

[N3559] Faisal Vali, Herb Sutter, Dave Abrahams, Proposal for Generic (Polymorphic) Lambda Expressions (Revision 2)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3559.pdf