Default constructible stateless lambdas

Document #:	D0624R0
Date:	2017-03-03
Project:	Programming Language C++
Audience:	Evolution Working Group

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

1 Introduction

Lambda expressions are a very convenient way of creating inline function objects, especially for passing to higher order algorithms. Furthermore, the Standard treats stateless lambdas specially, giving them an implicit conversion to a corresponding function pointer, which is incredibly useful for storing these in data structures. Unfortunately, stateless lambdas are not default constructible, which precludes many interesting use cases. This paper proposes making stateless lambdas default constructible, so that code like this is valid:

```
// in library.hpp
auto greater = [](auto x, auto y) { return x > y; };

// in user.cpp
// No need to care whether ‘greater‘ is a lambda or a function object
std::map<std::string, int, decltype(greater)> map;
```

2 Motivation

The first argument for this change is consistency. Indeed, since a lambda is just syntactic sugar for a function object, it makes sense that the two can be used interchangeably as much as possible:

```
auto greater = [](auto x, auto y) { return x > y; };
std::map<std::string, int, decltype(greater)> map;

// should be the same as

struct {
  template <typename X, typename Y>
  auto operator()(X x, Y y) const { return x > y; }
} greater;
std::map<std::string, int, decltype(greater)> map;
```

Instead, the former produces an error message saying something about the use of the deleted default constructor of the lambda used inside the map.

Apart from consistency, the second argument is that default-constructing stateless lambdas opens a very interesting design space for writing libraries. An important tenet of C++ is to give power to experts (while keeping simple things simple), and this tiny feature opens up a whole new class of use cases. For example, default-constructible lambdas allow the [Type Erased] library to implement runtime polymorphism with value semantics (like `std::function`, but for arbitrary interfaces) at the library level:

```
// Define the interface of something that can be drawn
struct Drawable : decltype(te::requires(
  "draw"_s = te::function<void (te::T const&, std::ostream&)>
)) { };

// Define how a type can satisfy the above concept
template <typename T>
auto const te::concept_map<Drawable, T> = te::make_concept_map(
  "draw"_s = [](T const& self, std::ostream& out) { self.draw(out); }
);

// Define an object that can hold anything that can be drawn
struct drawable {
  // ... boilerplate that would go away with reflection ...
};

struct Square {
  void draw(std::ostream& out) const { out << "Square"; }
};

struct Circle {
  void draw(std::ostream& out) const { out << "Circle"; }
};

void f(drawable d) {
  d.draw(std::cout);
}

// drawable is like std::function, but for Drawables instead of Callables
f(Square{...});
f(Circle{...});
```

While both the purpose of the library and its implementation are completely beyond the scope of this paper, the idea is that we need to transport the type of the lambda around without an instance of it, and then default-construct that lambda to fill an entry in the virtual table of an interface. This is one use case, and there are possibly many more to come, that currently require a gruesome hack to work (see this Gist).

3 Proposed Wording

The wording is based on the working paper [N4606]. At the beginning of **[expr.prim.lambda] 5.1.2/6**:

> The closure type for a lambda-expression with no lambda-capture has a public non-virtual defaulted default constructor. The closure type for a non-generic lambda-expression with no lambda-capture has a public non-virtual non-explicit const conversion function to pointer to function with C++ language linkage [...]

4 References

[Type Erased] Louis Dionne, *Type Erased: Runtime polymorphism done right*
https://github.com/ldionne/te

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