Default constructible and assignable stateless lambdas

Document #:	D0624R2
Date:	2017-11-09
Project:	Programming Language C++
Audience:	Core Working Group
Reply-to:	Louis Dionne <ldionne.2@gmail.com>

1 Revision history

- R0 – Initial draft
- R1
  - Rebase wording on top of the current working draft N4659.
  - Make stateless lambdas assignable per EWG direction in Toronto.
- R2
  - Rebase wording on top of current working draft [N4700] (no-op).
  - Fix wording to keep the assignment operator deleted for stateful lambdas.
  - Record the intent that construction and assignment operations be trivial.

2 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 neither default constructible nor assignable, which precludes many interesting use cases. This paper proposes making stateless lambdas default constructible and assignable, 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;
```

3 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:

```cpp
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. Furthermore, since lambdas are currently not assignable, the following would also produce an error:

```cpp
auto greater = [](auto x, auto y) { return x > y; };
std::map<std::string, int, decltype(greater)> map1{...}, map2{...};
map1 = map2; // oops!
```

Apart from consistency, the second argument is that making stateless lambdas more like normal function objects 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 [Dyno] library to implement runtime polymorphism with value semantics (like `std::function`, but for arbitrary interfaces) at the library level:

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

// Define how a type can satisfy the above concept
template <typename T>
auto const dyno::concept_map<Drawable, T> = dyno::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).

4 Proposed Wording

The wording is based on the working paper [N4700]. Replace in **[expr.prim.lambda] 8.1.5.1/11**:

> ~~The closure type associated with a *lambda-expression* has no default constructor and a deleted copy assignment operator.~~ The closure type associated with a *lambda-expression* has a deleted default constructor if the *lambda-expression* has a *lambda-capture*, and a defaulted default constructor otherwise. It has a deleted copy assignment operator if the *lambda-expression* has a *lambda-capture*, and a defaulted copy assignment operator otherwise. It has a defaulted copy constructor and a defaulted move constructor (15.8). [*Note:* These special member functions are implicitly defined as usual, and might therefore be defined as deleted. — *end note*]

When discussed in EWG, interest in having stateless lambdas be *trivially* default constructible was expressed, if CWG thought it to be a good idea. The current wording may not achieve that, so minor tweaking might be necessary if trivial default constructibility is desired.

5 References

[Dyno] Louis Dionne, *Dyno: Runtime polymorphism done right*
https://github.com/ldionne/dyno

[N4700] Richard Smith, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4700.pdf