Wording for Constexpr Lambda

Document #: P0170R1
Date : 2016-03-01
Revises : P0170R0
Working Group: Core
Reply to: Faisal Vali (faisalv@yahoo.com)

Faisal Vali
Jens Maurer
Richard Smith

Abstract

This paper presents core wording for the proposal N4487 that was accepted by the Evolution Working Group in Kona on 2015-10-22. N4487 proposed allowing certain *lambda-expressions* and operations on certain closure objects to appear within constant expressions. In doing so, N4487 proposed that a closure type be considered a literal type if the type of each of its data-members is a literal type; and, that if the `constexpr` specifier is omitted within the *lambda-declarator*, that the generated function call operator be `constexpr` if it would satisfy the requirements of a `constexpr` function (similar to the `constexpr` inference that already occurs for implicitly defined constructors and the assignment operator functions).

1 Précis

In brief, N4487 proposed the following:

1) *lambda-expressions* should be allowed to appear within constant expressions if the initialization of each of its closure-type's data members are allowed within a constant expression:

```
constexpr int AddEleven(int n) {
  // Initialization of the 'data member' for n can
  // occur within a constant expression since 'n' is
  // of literal type.
  return [n] { return n + 11; }();
}
static_assert(AddEleven(5) == 16, "");
```

2) The closure type should be a literal type if the type of each of its data-members is a literal type. This would allow the relevant special member functions to be `constexpr` (if not deleted) and thus evaluatable within constant expressions:

```
constexpr auto add = [] (int n, int m) {
  auto L = [=] { return n; };
  auto R = [=] { return m; };
  return [=] { return L() + R(); };
};
static_assert(add(3, 4)() == 7, "");
```

3) The `constexpr` specifier should be allowed within the *lambda-declarator* to specify the function call operator (or template) as `constexpr`:

```
auto ID = [] (int n) constexpr { return n; };
constexpr int I = ID(3);
```

4) If the `constexpr` specifier is omitted within the *lambda-declarator*, the function call operator (or template) is `constexpr` if it would satisfy the requirements of a `constexpr` function:

```
auto ID = [](int n) { return n; };
constexpr int I = ID(3);
```

5) The conversion function (to pointer-to-function) should, if it exists, be `constexpr`. If the corresponding function call operator is `constexpr`, the conversion function shall return the address of a function that is `constexpr`:

```
auto addOne = [] (int n) {
   return n + 1;
};
constexpr int (*addOneFp)(int) = addOne;
static_assert(addOneFp(3) == addOne(3), "");
```

2 Core Wording

In [basic.types] 3.9 change bullet 10.5.2:

A type is a literal type if it is:
(10.1) — possibly cv-qualified void; or
(10.2) — a scalar type; or
(10.3) — a reference type; or
(10.4) — an array of literal type; or
(10.5) — a possibly cv-qualified class type (Clause 9) that has all of the following properties:

(10.5.1) — it has a trivial destructor,
(10.5.2) — it is either a closure type (5.1.2 expr.prim.lambda), an aggregate type, (8.5.1) or has at least one constexpr constructor or constructor template that is not a copy or move constructor, and
(10.5.3) — all of its non-static data members and base classes are of non-volatile literal types.

In [expr.prim.lambda] 5.1.2/1 replace the `mutable`$_{opt}$ terminal with the *decl-specifier-seq*$_{opt}$ production, with the contraint that it shall only be mutable or constexpr

lambda-declarator:
(*parameter-declaration-clause*) ~~`mutable`$_{opt}$~~ *decl-specifier-seq*$_{opt}$
exception-specification$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$

In the *decl-specifier-seq* of the *lambda-declarator,* each *decl-specifier* shall either be `mutable` or `constexpr`.
[*Example:*

```
  auto monoid = [](auto v) { return [=] { return v; }; };

  auto add = [](auto m1) constexpr {
    auto ret = m1();
    return [=](auto m2) mutable {
        auto m1val = m1();
        auto plus = [=] (auto m2val) mutable constexpr
                        { return m1val += m2val; };
        ret = plus(m2());
        return monoid(ret);
    };
  };
```

```
constexpr auto zero = monoid(0);
constexpr auto one = monoid(1);
static_assert(add(one)(zero)() == one());  // OK
```

// Since 'two' below is not declared constexpr, an evaluation of its constexpr member function call operator
// can not perform an lvalue-to-rvalue conversion on one of its subobjects (that represents its capture)
// in a constant expression.

```
auto two = monoid(2);
assert(two() == 2); // OK, not a constant expression.
static_assert(add(one)(one)() == two()); // ill-formed: two() is not a constant expression
static_assert(add(one)(one)() == monoid(2)()); // OK
```

— *end example*]

Change [expr.prim.lambda] 5.1.2/3

The type of the lambda-expression (which is also the type of the closure object) is a unique, unnamed nonunion class type — called the closure type — whose properties are described below. This class type is ~~neither an aggregate (8.5.1) nor a literal type (3.9)~~ not an aggregate type (8.5.1). …

Change [expr.prim.lambda] 5.1.2/5:

… This function call operator or operator template is declared `const` (9.3.1) if and only if the *lambda-expression's parameter-declaration-clause* is not followed by `mutable`. It is neither virtual nor declared `volatile`. Any *exception-specification* specified on a *lambda-expression* applies to the corresponding function call operator or operator template. An *attribute-specifier-seq* in a *lambda-declarator* appertains to the type of the corresponding function call operator or operator template. The function call operator or any given operator template specialization is a constexpr function if either the corresponding *lambda-expression's parameter-declaration-clause* is followed by `constexpr`, or it satisfies the requirements for a constexpr function (7.1.5). [Note: Names referenced in the lambda-declarator are looked up in the context in which the lambda-expression appears. —end note]

[*Example:*

```
auto ID = [](auto a) { return a; };
static_assert(ID(3) == 3); // OK

struct NonLiteral {
  NonLiteral(int n) : n(n) { }
  int n;
};
```

```
static_assert(ID(NonLiteral{3}).n == 3); // ill-formed
```

— *end example*]

Change [expr.prim.lambda] 5.1.2/6

The closure type for a non-generic *lambda-expression* with no *lambda-capture* has a public `constexpr` non-virtual non-explicit const conversion function to pointer to function with C++ language linkage (7.5) having the same parameter and return types as the closure type's function call operator. The value returned by this conversion function is ~~shall be~~ the address of a function `F` that, when invoked, has the same effect as invoking the closure type's function call operator. `F` is a constexpr function if the function call operator is a constexpr function. For a generic lambda with no lambda-capture, the closure type has a public `constexpr` non-virtual non-explicit const conversion function template to pointer to function. …

The value returned by any given specialization of this conversion function template is ~~shall be~~ the address of a function `F` that, when invoked, has the same effect as invoking the generic lambda's corresponding function call operator template specialization. `F` is a constexpr function if the corresponding specialization is a constexpr function.[Note: ...

[*Example:*

```
auto Fwd = [](int (*fp)(int), auto a) { return fp(a); };
auto C = [](auto a) { return a; };

static_assert(Fwd(C,3) == 3); // OK

// No specialization of the function call operator template can  be constexpr (because of the local static).
auto NC = [](auto a) { static int s; return a; };
static_assert(Fwd(NC,3) == 3); // ill-formed
```

— *end example*

Change [expr.prim.lambda] 5.1.2/16:

An entity is captured by reference if it is implicitly or explicitly captured but not captured by copy. It is unspecified whether additional unnamed non-static data members are declared in the closure type for entities captured by reference. If declared, such non-static data members shall be of literal type.

[*Example:*

```
// The inner closure type must be a literal type regardless of how reference captures are represented.
static_assert([](int n) { return [&n] { return ++n; }(); }(3) == 4);
```

— *end example*]

Remove bullet [expr.const] 5.20/2.6:

~~— a lambda-expression (5.1.2);~~

Modify bullet [expr.const] 5.20/2.10:

— in a lambda-expression, a reference to `this` or to a variable with automatic storage duration defined outside that lambda-expression, where the reference would be an odr-use (3.2, 5.1.2); [*Example:*

```
void g() {
  const int n = 0;
  [=] {
    constexpr int i = n;    // OK, 'n' is not odr-used and not captured here.
    constexpr int j = *&n;  // Ill-formed, '&n' would be an odr-use of 'n'.
  };
}
```

— end example]

[*Note*: If the odr-use occurs in an invocation of a function call operator of a closure type, it no longer refers to `this` or to an enclosing automatic variable due to the transformation (5.1.2) of the *id-expression* into an access of the corresponding data member *— end note*][*Example:*

```
// 'v' & 'm' are odr-used but do not occur in a constant-expression within the nested
// lambda, so are well-formed.
auto monad = [](auto v) { return [=] { return v; }; };
auto bind = [](auto m) {
        return [=](auto fvm) { return fvm(m()); };
};

// OK to have captures to automatic objects created during constant expression evaluation.
static_assert(bind(monad(2))(monad)() == monad(2)());
```

— end example]

Modify 7.1/2:

Each decl-specifier shall appear at most once in a ~~the~~ complete *decl-specifier-seq* ~~of a declaration~~, except that long may appear twice.

3 Acknowledgment

Ville Voutilainen & Gabriel Dos Reis for co-authoring the original proposal: N4487