N2859=09-0049

2009-03-19

Daveed Vandevoorde (daveed@edg.com)

# New wording for C++0x Lambdas

## Introduction

During the meeting of March 2009 in Summit, a large number of issues relating to C++0x Lambdas were raised and reviewed by the core working group (CWG). After deciding on a clear direction for most of these issues, CWG concluded that it was preferable to rewrite the section on Lambdas to implement that direction. This paper presents this rewrite.

## Open issue

There are known problems with move constructors that might have to deal with an exception after some subobjects have already been moved. When that general issue is addressed, the move constructor for closure types will likely require some treatment.

## Resolved issues

The following CWG issues are addressed by this rewrite:

680: What is a move constructor?
720: Need examples of *lambda-expressions*
750: Implementation constraints on reference-only closure objects
751: Deriving from closure classes
752: Name lookup in nested *lambda-expressions*
753: Array names in lambda capture sets
754: Lambda expressions in default arguments of block-scope function declarations
756: Dropping cv-qualification on members of closure objects
759: Destruction of closure objects
761: Inferred return type of closure object call operator
762: Name lookup in the compound-statement of a lambda-expression
763: Is a closure object's **`operator()`** inline?
764: Capturing unused variables in a lambda expression
766: Where may lambda expressions appear?
767: **`void`** parameter for lambdas
768: Ellipsis in a lambda parameter list
769: Initialization of closure objects
771: Move-construction of reference members of closure objects
772: *capture-default* in lambdas in local default arguments
774: Can a closure class be a POD?
775: Capturing references to functions
779: Rvalue reference members of closure objects?
782: Lambda expressions and argument-dependent lookup

In addition, this rewrite adds the restriction that lambda expressions cannot be used in the operand of a **`sizeof`** operator, **`alignof`** operator, or **`decltype`** specifier. That restriction—suggested by Doug Gregor and John Spicer—avoids severe implementation difficulties with template argument deduction (e.g., this avoids the need to encode arbitrary statement sequences in mangled names).

## Key concepts in the new wording

The new wording no longer relies on lookup to remap uses of captured entities. It more clearly denies the interpretations that a lambda's *compound-statement* is processed in two passes or that any names in that *compound-statement* might resolve to a member of the closure type.

The new wording no longer specifies any rewrite or closure members for "by reference" capture. Uses of entities captured "by reference" affect the original entities, and the mechanism to achieve this is left entirely to the implementation.

The term "*late-specified return type*" has been dropped in favor of a nonterminal *trailing-return-type*.

## Wording changes

The following changes are relative to N2798.

In 3.3.2 [basic.scope.local] paragraph 2 replace *lambda-parameter-declaration-clause* by *lambda-declarator* (one occurrence).

Replace subsection 5.1.1 [expr.prim.lambda] by the following:

### 5.1.1 Lambda expressions [expr.prim.lambda]

1 Lambda expressions provide a concise way to create simple function objects. [ *Example:*

```
#include <algorithm>
#include <cmath>
void abssort(float *x, unsigned N) {
  std::sort(x, x+N,
            [](float a, float b) {
              return std::abs(a) < std::abs(b);
            });
}
```

—*end example* ]

*lambda-expression:*
  *lambda-introducer lambda-declarator*$_{opt}$ *compound-statement*

*lambda-introducer:*
  **[** *lambda-capture*$_{opt}$ **]**

*lambda-capture:*
  *capture-default*
  *capture-list*

*capture-default* **,** *capture-list*

*capture-default:*
**&**
**=**

*capture-list:*
*capture*
*capture-list* **,** *capture*

*capture:*
*identifier*
**&** *identifier*
**this**

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

2 The evaluation of a *lambda-expression* results in an rvalue temporary (_class.temporary_ 12.2). This temporary is called the *closure object*. A *lambda-expression* shall not appear in an unevaluated operand (_expr_ Clause 5). [ *Note:* A closure object behaves like a function object (_function.objects_ 20.7). —*end note* ]

3 The type of the *lambda-expression* (which is also the type of the closure object) is a unique, unnamed non-union class type—called the *closure type*—whose properties are described below. The closure type is declared in the smallest block scope, class scope, or namespace scope that contains the associated *lambda-expression*. [ *Note:* This determines the set of namespaces and classes associated with the closure type (_basic.lookup.argdep_ 3.4.2). —*end note* ] An implementation may define the closure type differently from what is described below provided this does not alter the observable behavior of the program other than by changing:

— the size and/or alignment of the closure type
— whether the closure type is trivially copyable (_class_ Clause 9)
— whether the closure type is a standard-layout class (_class_ Clause 9)
— whether the closure type is a POD class (_class_ Clause 9)

4 If a *lambda-expression* does not include a *lambda-declarator*, it is as if the *lambda-declarator* were **()**. If a *lambda-expression* does not include a *trailing-return-type*, it is as if the *trailing-return-type* denotes the following type:

— if the *compound-statement* is of the form
**{ return** *attribute-specifier*$_{opt}$ *expression* **; }**
the type of the returned expression after lvalue-to-rvalue conversion (_conv.lval_ 4.1), array-to-pointer conversion (_conv.array_ 4.2), and function-to-pointer conversion (_conv.func_ 4.3);
— otherwise, **void**.

[ *Example:*

```
auto x1 = [](int i){ return i; };
  // OK: return type is int
auto x2 = []{ return { 1, 2 }; };
  // error: the return type is void (a braced-init-list is not an expression)
```

—*end example* ]

5 The closure type for a *lambda-expression* has a public **inline** function call operator (_over.call_ 13.5.4) whose parameters and return type are described by the *lambda-expression*'s *parameter-declaration-clause* and *trailing-return-type* respectively. This function call operator is declared **const** (_class.mfct.non-static_ 9.3.1) if and only if the *lambda-expression*'s *parameter-declaration-clause* is not followed by **mutable**. It is not declared **volatile**. Default arguments (_decl.fct.default_ 8.3.6) shall not be specified in the *parameter-declaration-clause* of a *lambda-declarator*. Any *exception-specification* specified on a *lambda-expression* is that of the corresponding function call operator. Any *attribute-specifier*s appearing immediately after the *lambda-expression*'s *parameter-declaration-clause* appertain to the type of the corresponding function call operator.

6 The *lambda-expression*'s *compound-statement* yields the *function-body* (_dcl.fct.def_ 8.4) of the function call operator, but for purposes of name lookup (_basic.lookup_ 3.4), determining the type and value of **this** (_class.this_ 9.3.2), and transforming *id-expressions* referring to non-static class members into class member access expressions using **(*this)** (_class.mfct.non-static_ 9.3.1), the *compound-statement* is considered in the context of the *lambda-expression*. [ *Example:*

```
struct S1 {
  int x, y;
  int operator()(int);
  void f() {
    [=]()->int {
      return operator()(this->x+y);
        // equivalent to: S1::operator()(this->x+(*this).y)
        //   and this has type S1*
    };
  }
};
```

—*end example* ]

7 For the purpose of describing the behavior of *lambda-expression*s below, **this** is considered to be "used" if replacing **this** by an invented variable **v** with automatic storage duration and the same type as **this** would result in **v** being used (_basic.def.odr_ 3.2).

8 If a *lambda-capture* includes a *capture-default* that is **&**, the identifiers in the *lambda-capture* shall not be preceded by **&**. If a *lambda-capture* includes a *capture-default* that is **=**, the *lambda-capture* shall not contain **this** and each identifier it contains shall be preceded by **&**. An *identifier* or **this** shall not appear more than once in a *lambda-capture*. [ *Example:*

```
struct S2 { void f(int i); };
void S2::f(int i) {
  [&, i]{};        // OK
  [&, &i]{};       // error: i preceded by & when & is the default
  [=, this]{};     // error: this when = is the default
  [i, i]{};        // error: i is repeated
}
```

—*end example* ]

9 A *lambda-expression*'s *compound-statement* can use (see above) **this** from an immediately-enclosing member function definition, as well as variables and references with automatic storage duration from an immediately-enclosing function definition or *lambda-expression*, provided these entities are captured (as described below). Any other use of a variable or reference with automatic storage duration declared outside the *lambda-expression* is ill-formed. [ *Example:*

```
void f1(int i) {
  int const N = 20;
  [=]{
    int const M = 30;
    [=]{
      int x[N][M]; // OK: N and M are not "used"
      x[0][0] = i; // error: i is not declared in the immediately
    };             //        enclosing lambda-expression
  };
}
```

—*end example* ]

10 The *identifier*s in a *capture-list* are looked up using the usual rules for unqualified name lookup (_basic.lookup.unqual_ 3.4.1); each such lookup shall find a variable or reference with automatic storage duration. Entities (variables, references, or **this**) appearing in the *lambda-expression*'s *capture-list* are said to be *explicitly captured*.

11 If a *lambda-expression* has an associated *capture-default* and its *compound-statement* uses (_basic.def.odr_ 3.2) **this** or a variable or reference with automatic storage duration declared in an enclosing function or *lambda-expression* and the used entity is not explicitly captured, then the used entity is said to be *implicitly captured*. [ *Note:* Implicit uses of **this** can result in implicit capture. —*end note* ]

12 If **this** is captured, either explicitly or implicitly, the *lambda-expression* shall appear directly in the definition of a non-static member function, i.e., not in another *lambda-expression*. [*Note:* This rule prevents access from a nested *lambda-expression* to the members of the enclosing *lambda-expression*'s closure object. —*end note* ]

13 A *lambda-expression* appearing in a default argument shall not implicitly or explicitly capture any entity. [ *Example:*

```
void f2() {
  int i = 1;
  void g1(int = ([i]{ return i; })());        // ill-formed
```

```
    void g2(int = ([i]{ return 0; })());         // ill-formed
    void g3(int = ([=]{ return i; })());         // ill-formed
    void g4(int = ([=]{ return 0; })());         // OK
    void g5(int = ([]{ return sizeof i; })());   // OK
  }
```

—*end example* ]

14 An entity is *captured by copy* if it is implicitly captured and the *capture-default* is **=**, or if it is explicitly captured with a *capture* that does not include a **&**. For each entity captured by copy, an unnamed non-static data member is declared in the closure type. The declaration order of these members is unspecified. The type of such a data member is the type of the corresponding captured entity if the entity is not a reference to an object, or the referenced type otherwise. [ *Note:* If the captured entity is a reference to a function, the corresponding data member is also a reference to a function. —*end note* ]

15 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.

16 Every *id-expression* that is a use (_basic.def.odr_ 3.2) of an entity captured by copy is transformed into an access to the corresponding unnamed data member of the closure type. If **this** is captured, each use of **this** is transformed into an access to the corresponding unnamed data member of the closure type cast (_expr.cast_ 5.4) to the type of **this**. [ *Note:* The cast ensures that the transformed expression is an rvalue. —*end note* ]

17 Every occurrence of **decltype((***x***))** where *x* is a possibly parenthesized *id-expression* that names an entity of automatic storage duration is treated as if *x* were transformed into an access to a corresponding data member of the closure type that would have been declared if *x* were a use of the denoted entity. [ *Example:*

```
  void f3() {
    float x, &r;
    [=]{                        // x and r are not captured (appearance in a
                                // decltype operand is not a "use")
      decltype(x) y1;           // y1 has type float
      decltype((x)) y2 = y1;    // y2 has type float const& because this
                                //   lambda is not mutable and x is an lvalue
                                //   even after the hypothetical transformation
      decltype(r) r1 = y1;      // r1 has type float& (transformation not
                                //   considered)
      decltype((r)) r2 = y2;    // r2 has type float const&
    };
  }
```

—*end example* ]

18 The closure type associated with a *lambda-expression* has a deleted default constructor and a deleted copy assignment operator. It has an implicitly-declared copy constructor

(_class.copy_ 12.8). [ *Note:* The copy constructor is implicitly defined in the same way as any other implicitly declared copy constructor would be implicitly defined. —*end note* ]

19 The closure type *C* associated with a *lambda-expression* has an additional public **inline** constructor with a single parameter of type *C*&&. Given an argument object *x*, this constructor direct-initializes each non-static data member *m* of ***this** with an expression equivalent to **std::move(***x.m***)** if *m* is not a reference, or with *x.m* if *m* is a reference. [ *Note:* The notations are for exposition only; the members of a closure type are unnamed and **std::move** need not be called. —*end note* ]

20 The closure type associated with a *lambda-expression* has an implicitly-declared destructor (_class.dtor_ 12.4).

21 When the *lambda-expression* is evaluated, the entities that are captured by copy are used to direct-initialize each corresponding non-static data member of the resulting closure object. (For array members, the array elements are direct-initialized in increasing subscript order.) These initializations are performed in the (unspecified) order in which the non-static data members are declared. [ *Note:* This ensures that the destructions will occur in the reverse order of the constructions. —*end note* ]

22 [ *Note:* If an entity is implicitly or explicitly captured by reference, invoking the function call operator of the corresponding *lambda-expression* after the lifetime of the entity has ended is likely to result in undefined behavior. —*end note* ]

In 7.1.6.4 [dcl.spec.auto] replace the introductory paragraphs

The **auto** *type-specifier* signifies that the type of an object being declared shall be deduced from its initializer or specified explicitly at the end of a function declarator.

The **auto** *type-specifier* may appear with a function declarator with a late-specified return type (8.3.5) in any context where such a declarator is valid, and the use of **auto** is replaced by the type specified at the end of the declarator.

by

The **auto** *type-specifier* signifies that the type of a variable or reference being declared shall be deduced from its initializer or that a function declarator shall include a *trailing-return-type*.

The **auto** *type-specifier* may appear with a function declarator with a *trailing-return-type* (_dcl.fct_ 8.3.5) in any context where such a declarator is valid.

In 8 [dcl.decl] paragraph 4 replace the grammar line

*noptr-declarator parameters-and-qualifiers* **->** *attribute-specifier$_{opt}$ type-id*

by

*noptr-declarator parameters-and-qualifiers trailing-return-type*

and add before the grammar rule for *ptr-operator*:

*trailing-return-type:*

**->** *attribute-specifier*$_{opt}$ *type-id*

In 8.1 [dcl.name] paragraph 1 replace the grammar line

*noptr-abstract-declarator*$_{opt}$ *parameters-and-qualifiers*
**->** *attribute-specifier*$_{opt}$ *type-id*

by

*noptr-abstract-declarator*$_{opt}$ *parameters-and-qualifiers* *trailing-return-type*

Add a new paragraph with the following content at the end of 8 [dcl.decl]:

5 The optional *attribute-specifier* in a *trailing-return-type* appertains to the indicated return type. The *type-id* in a *trailing-return-type* includes the longest possible sequence of *abstract-declarators*. [ *Note:* This resolves the ambiguous binding of array and function declarators. [ *Example:*

```
auto f()->int(*)[4]; // function returning a pointer to array[4] of int
                     // not function returning array[4] of pointer to int
```

*—end example* ] *—end note* ]

In 8.3.5 [dcl.fct] paragraph 2 replace the grammatical form

**D1 (** *parameter-declaration-clause* **)** *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$
*ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ **->** *attribute-specifier*$_{opt}$ *type-id*

by

**D1 (** *parameter-declaration-clause* **)** *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$
*ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ *trailing-return-type*

and replace the sentences

Such a function type has a *late-specified return type*. The first optional *attribute-specifier* appertains to the function type. The second optional *attribute-specifier* appertains to the return type.

by

The optional *attribute-specifier* appertains to the function type.

In 8.3.5 [dcl.fct] delete paragraph 3.

In the last note of 8.3.5 [dcl.fct] paragraph 12 replace late-specified return type by *trailing-return-type* (two occurrences: one plural and one singular).