Constness of Lambda Functions (Revision 1)

Document no: N2658=08-0168

Jaakko Järvi* Peter Dimov† John Freeman‡

2008-06-12

1 Introduction

Lambda expressions, as specified in the document N2550 [JFC08b], were voted into the working paper in the Bellevue meeting in March 2008. As a result of discussions in the evolution and core working groups, N2550 introduced a change over the earlier proposal N2529 [JFC08a] in how constness of a closure object affects the constness of the variables stored in the closure. The document N2651 [JD08] revisited that decision, and suggested a small change to the specification of lambda expressions in this aspect. This paper revises N2651, and reflects the decisions made in the Core Working Group in the Sophia-Antipolis meeting. The discussion of alternatives from N2651 is not included—we only describe the design agreed upon, and necessary wording changes against the current working paper N2606.

2 Background

The result of evaluating a lambda expression is a closure object. A closure object can store copies of variables defined in the enclosing scope of the lambda expression as its member variables. Closure objects behave as function objects where, according to the current specification in the working paper, the function call operator is defined to be **const**. This allows invocation of a closure object regardless of whether the object is **const** or not, but prevents modifying the closure members in the body of the lambda expression. This is somewhat limiting. For example, the following example is ill-formed, as acc is effectively **const** in the lambda expression.

```
vector<int> a;
  ...
int acc = 0;
transform(a.begin(), a.end(), a.begin(), [acc](int x) { return acc += x; });
```

The following function object is comparable to the object constructed from the above lambda expression:

```
class A {
  int acc;
public:
  // constructor
  int operator()(int x) const { return acc += x; }
}
```

One can work around the constness by storing the state outside of the closure object:

```
transform(a.begin(), a.end(), a.begin(), [&acc](int x) { return acc += x; });
```

*jarvi@cs.tamu.edu
†pdimov@mmltd.net
‡jfreeman@cs.tamu.edu

This is often undesirable. For example, in above code, the change of value of acc may or may not be an expected side effect. In particular, if closures are invoked in concurrent threads, these kind of aliasing issues and side-effects complicate reasoning about programs.

3 Proposal

We propose that the syntax of the lambda expressions be extended to allow qualification with the **mutable** keyword. The place for this qualification should be between the parameter list of the lambda expression and the optional exception specification.

The absence or, respectively, presence of the **mutable**-qualification would determine whether the function call operator of the closure object is const or, respectively, non-const. We note that the semantics of closure objects without the **mutable** keyword remains unchanged from the semantics specified in the working paper.

Examples:

```
int x;
[x]() { ++x; } // error, x is const
[x]() mutable { ++x; } // OK

template <class F> void by_copy(F f) { f(); }
template <class F> void by_const_reference(const F& f) { f(); }

by_copy([]() mutable {}); // OK
by_copy([]() {}); // OK

by_const_reference([]() mutable {}); // error, calling a non−const member function of a const object
by_const_reference([]() {}); // OK
```

References

[JD08] Jaakko Järvi and Peter Dimov. Constness of lambda functions. Technical Report N2651=08-0161, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, May 2008.

[JFC08a] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda expressions and closures: Wording for monomorphic lambdas (revision 3). Technical Report N2529=08-0039, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2008.

[JFC08b] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda expressions and closures: Wording for monomorphic lambdas (revision 4). Technical Report N2550=08-0060, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2008.

Proposed Wording

5.1.1 Lambda Expressions [expr.prim.lambda]

lambda-parameter-declaration:
`(` *lambda-parameter-declaration-list*$_{opt}$ `)` `mutable`$_{opt}$ *exception-specification*$_{opt}$ *lambda-return-type-clause*$_{opt}$

9 *F* has a public ~~`const`~~ function call operator ([over.call], 13.5.4) with the following properties:

- The *parameter-declaration-clause* is the *lambda-parameter-declaration-list*.
- The return type is the type denoted by the *type-id* in the *lambda-return-type-clause*; for a lambda expression that does not contain a *lambda-return-type-clause* the return type is `void`, unless the *compound-statement* is of the form `{ return` *expression*`; }`, in which case the return type is the type of *expression*.
- The *cv-qualifier-seq* is absent if the lambda expression is `mutable`, and it is `const` otherwise.
- The *exception-specification* is the lambda expression's *exception-specification*, if any.
- The *compound-statement* is obtained from the lambda expression's *compound-statement* as follows: If the lambda expression is within a non-static member function of some class *X*, transform *id-expression*s to class member access syntax as specified in ([class.mfct.non-static], 9.3.1), then replace all occurrences of `this` by *t*. [*Note:* References to captured variables or references within the *compound-statement* refer to the data members of *F*. — *end note*]

11 If every name in the effective capture set is preceded by `&` and the lambda expression is not `mutable`, *F* is publicly derived from `std::reference_closure<R(P)>` ([func.referenceclosure], 20.5.17), where `R` is the return type and `P` is the parameter-type-list of the lambda expression. Converting an object of type *F* to type `std::reference_closure<R(P)>` and invoking its function call operator shall have the same effect as invoking the function call operator of *F*.