Constness of Lambda Functions

Document no: N2651=08-0161

Jaakko Järvi* Peter Dimov†

2008-05-19

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. This paper revisits that decision, and suggest a small change to the specification of lambda expressions in this aspect.

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 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(), [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; }
}
```

In versions of the lambda proposal prior to N2550, non-reference closure members, such as acc above, were declared **mutable**, so that, e.g., the code above would have been well-defined. Lambda functions thus had no notion of "constness".

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

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

*jarvi@cs.tamu.edu
†pdimov@mmltd.net

This is undesirable. For example, in above code, the change of value of acc may or may not be an expected side effect. In particular, if the closure is invoked in a concurrent thread, these kind of side-effects complicate reasoning about the program.

Several suggestions (other than those in N2550 and N2529) have been made in the design space of lambdas' constness, including:

- A closure object should define two function call operators, one **const** and the other non-**const** (or four, for all combinations of **const** and **volatile**, and rvalue references could be added to the mix as well)

  The downside of this approach is that the same code needs to be generated multiple times, of which typically only one would be used, thus bloating the size of executables.

- The syntax for lambda expressions could be extended to allow declaring whether the closure members should be declared **mutable** or not.

  This approach could be confusing to programmers, as the mutability is not a property of the closure object, but rather the variables stored in the closure.

- The syntax for lambda expressions could be extended to allow declaring whether the function call operator should be **const** or not.

Of these, we propose a change according to the last item—actually proposing to allow arbitrary *cv*-qualification of lambda expressions for uniformity, even though we do not see many uses for **volatile** closure objects.

3 Proposal

We propose that the syntax of the lambda expressions be extended to allow *cv*-qualifiers. The *cv*-qualification would determine the *cv*-qualification of the function call operator in the closure object. The place for the *cv*-qualifiers should be between the parameter list of the lambda expression and the optional exception specification. We note that the semantics of closure objects as specified in the working paper coincides with that of a closure object constructed from a **const** lambda expression as proposed here.

We note that the common idiom, e.g., followed by the standard library, is to pass function objects by copy. Using this idiom, a non-const closure object works perfectly well. The above example thus would work. If, however, a template function takes a function object argument by **const** reference (and invokes the function object), the lambda function must be declared const. For such a lambda function, closure objects cannot modify their non-reference members.

Examples:

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

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

by_copy([](){}); // ok
by_copy([]() const {}); // ok

by_const_reference([](){}); // error
by_const_reference([]() const {}); // ok
```

3.1 Discussion

The downside of the proposed mechanism is that we are imposing a syntactic "tax" on the typical use. In cases where a closure has no state, e.g., with capture lists like [], [&], and [&a, &b], requiring the **const** qualification to make the closure object callable in a "**const** context" may seem like unnecessary boilerplate.

One possibility to evade this boilerplate is to add a rule that the function call operator of closures with no state (i.e., that are of type std::reference_closure) is **const**.

Finally, we view generating both a **const** and non-**const** overload of the closure's function call operator as the best design from the usability and convenience point of view. In many cases (stateless closures and more), the two overloads will be identical and could be served by a single **const** overload, possibly mitigating the concerns of bloating executables enough to make the design a viable option.

References

[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*$_{opt}$) *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$ *lambda-return-type-clause*$_{opt}$

6 The type of the closure object is a class with a unique name, call it F, considered to be defined at the point where the lambda expression occurs.

Each name `N` in the effective capture set is looked up in the context where the lambda expression appears to determine its object type; in the case of a reference, the object type is the type to which the reference refers. For each element in the effective capture set, F has a private non-static data member as follows:

— if the element is `this`, the data member has some unique name, call it t, and is of the type of `this` ([class.this], 9.3.2);

— if the element is of the form `& N`, the data member has the name `N` and type "reference to object type of `N`";

— otherwise, the element is of the form `N`, the data member has the name `N` and type "cv-unqualified object type of `N`".

The declaration order of the data members is unspecified.

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 the lambda expression's *cv-qualifier-seq*, if any.

— 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*]