Doc No: WG21 N3424 = .16 12-0114
Date: 2012-09-23
Reply to: Herb Sutter (hsutter@microsoft.com)
Subgroup: EWG – Evolution

Lambda Correctness and Usability Issues

Herb Sutter

Lambda functions are a hit – they are wonderfully useful, and increasingly widely used, so that we are now gaining real-world experience with them in large code bases.

Practical field experience with lambdas has demonstrated two issues with existing lambdas[1] that are responsible for the majority of usability and correctness problems in practice. This paper summarizes these two issues, which are characterized by:

- they are sources of ongoing correctness problems (people accidentally writing bugs) and usability problems (programmer surprise); and
- they require some breaking changes[2] to fix what are arguably (important but fortunately small) design bugs.

This is a discussion paper that aims to expose the problems for EWG discussion in Portland.

Capturing *this*

"We use lambdas extensively in our very large code base. The vast majority of errors we encounter in practice are because of the implicit capture of this." – Customer

There is only one variable that can be captured without mentioning its name: *this*. This is causing ongoing confusion because:

[1] Not counting extension requests, such as move capture or generic/polymorphic lambdas, some of which are the subjects of other papers in this mailing.

[2] Breaking changes of the kind proposed in this paper can be dealt with via bumping *__cplusplus* in the standard and normal compiler migration techniques in implementations, such as has been used effectively in the past for breaking changes such as when we changed the lifetime for for-loop scoped variables: transition users by permitting the old semantics under a compatibility switch, and progress through the usual migration phases (e.g., warning-optional -> warn-by-default -> error-by-default -> unsupported) across multiple releases of the compiler so users can choose when to adapt. Note that it can be (and has been) argued that when we changed the lifetime of for-loop scope variables there technically was no standard and so we did not create an incompatibility or breaking change in the standard, and that doing so in a published standard is different, this argument is on a mere technicality: what matters most is the effect of breaking changes in practice on real user code, and in the case of the for-loop scope change the committee and the user community knew well that the change would and did in practice break a lot of code (far more than would be affected by the changes proposed herein), and viewed the breaking change as a positive thing.

- it captures something invisible, and that most programmers are not aware of because it relies on what is arguably an implementation detail of C++ name lookup (i.e., that in a member function the name of a member variable *m* is rewritten *this->m*);
- it causes member variables to be captured as if by reference, which is especially confusing in *[=]* lambdas where users think they said to explicitly capture everything by value; and
- in fact, member variables cannot be captured by value at all, but this is not evident in the syntax, which allows their capture via *this* and syntactically implies that they are captured by value.

For example, if *x* and *y* are local variables and *z* is a member variable:

```
void myclass::myfunc( int x ) {

   yeti y;

   auto lambda = [=] {
      f( x );              // x captured by value
      y.g();               // y captured by value
      cout << z;           // this captured by value if z is a member variable…
                           // … and z is captured as if by reference
   };

}
```

The mention of *z* compiles but does not cause capture of *z*(!), it actually captures *this* which is never mentioned, and it has the semantics of capturing *z* effectively by reference. (If *z* had been a local variable, it would have been captured by value. But because *z* happens to be a member variable, is it captured as if by reference.)

How extensive is this problem? I've encountered it myself many times and encountered many customers who've complained about it, but what prompted me to write this paper was two particular data points at *C++ and Beyond* in August 2012: a prominent instructor felt compelled to include these surprising semantics and workarounds on a list of first-order usability issues to teach people about in C++11; and one company attending the event reported that they use (and love) lambdas extensively in their large code base, empirically measured the sources of problems with lambdas, and have found that the vast majority of their lambda problems and bugs are due to this one problem. Further, it turns out that programmers do want to capture member variables by value, and currently have no direct way to do so.

Here are three possible resolutions, each of which would require a breaking change.

(Poor) Option 1: Require capture of *this* to be explicit. That is, *[=]* or *[&]* would not capture *this*, and the programmer would be required to write *[this]*. The above example would have to be written as:

```
// Option 1: Require [this]

void myclass::myfunc( int x ) {

   yeti y;

   auto lambda = [=, this] {
      f( x );              // x captured by value
```

```
        y.g();                      // y captured by value
        cout << z;                  // this captured by value if z is a member variable…
                                    // … and z is captured as if by reference
    };

}
```

This makes it slightly clearer to the programmer what's actually going on (he is told to add '*, this*', but the captured variable is still invisible at point of use). However, it arguably creates a new asymmetry ('the capture default applies to all variables, er, well all except one'), and it does not address by-value capture of members.

(Fair) Option 2: Require use of *this* to be explicit. That is,

```
// Option 2: Require this->

void myclass::myfunc( int x ) {

    yeti y;

    auto lambda = [=] {
        f( x );                     // x captured by value
        y.g();                      // y captured by value
        cout << this->z;            // this captured by value (clearer that z is captured as if by reference)
    };

}
```

This makes it clear to the programmer what's actually going on. However, it also creates an asymmetry (*this*-> is not otherwise required in member functions), and still does not address by-value capture of members.

(Good) Option 3: Add support for capture of member variables with the obvious semantics. Programmers already expect this, and are surprised when it does not happen. Even better, this can be viewed as a supserset of Option 2 – as in Option 2, the explicit *this*-> syntax is likewise supported and is the only way to capture *this*, and we are adding a pure extension to Option 2 in the capability of capturing the member variables directly as well.

In Option 3, the above example would have its semantics be changed to value capture, and explicit *this*-> capture is also allowed:

```
// Option 3: Support member variable capture  – note [this] is still allowed

void myclass::myfunc( int x ) {

    yeti y;

    auto lambda = [=] {
        f( x );                     // x captured by value
        y.g();                      // y captured by value
        cout << z;                  // z captured by value
```

```
            cout << this->z;      // this captured by value (clearer that z is captured as if by reference)
        };

    }
```

This makes it clear to the programmer what is going on, matches what experience has shown to be many programmers' expectation (namely that *z* be captured by value), and addresses by-value capture of members. However, it is still a breaking change because it changes the meaning of existing code that mentions member variables; existing code that uses as-if-by-reference capture of member variables would now get a by-value copy capture instead.

The best part of Option 3 is that it's what all programmers I've worked with actually think is supposed to happen when they first use the feature.

Breaking change impact assessment:

- All valid code that (a) captures by *[=]* default, and also (b) captures *this* implicitly by using a member variable name without *this->* in the lambda body, will now capture copies of member variables instead of references to them. In some cases this will fix bugs; but in cases where the by-reference capture was desired, the user should either capture the member variables by reference or else use *this->* explicitly in the lambda body.

Implementation efficiency note: If the user specifies capturing multiple member variables by reference, instead of storing one pointer/reference for each member, an implementation could still store only a single copy of the *this* pointer as an optimization.

The odd couple: Capture by value's injected *const* and quirky *mutable*

Consider this strawman example, where the programmer captures a local variable by value and tries to modify the captured value (which is a member variable of the lambda object):

```
int val = 0;

auto x = [=]( item e )                // look ma, [=] means explicit copy
            {  use( e, ++val );  };   // error: count is const, need 'mutable'

auto y = [val]( item e )              // darnit, I really can't get more explicit
            {  use( e, ++val );  };   // same error: count is const, need 'mutable'
```

This feature appears to have been added out of a concern that the user might not realize he got a copy, and in particular that since lambdas are copyable he might be changing a different lambda's copy.

However, the user explicitly asked for a copy, and in practice what has proven to be actually surprising to users is that the code is not allowed. Users we have observed expect the code to work and to modify the captured copy. Fortunately it doesn't come up often because typically lambdas don't try to use their copied captures as local variables, but when they users do want to do that for convenience they are regularly surprised that: (a) they can't modify the copy; and (b) what they have to write to make the code compile ("*mutable*? … what, really? … and where?").

Users tend to view this as nannying, and disallowing it is exactly the same as if we disallowed the following (which I suspect no one would tolerate):

```
void f( int count ) {                      // pass by value – explicit copy
   …
   do_something_with( ++count );           // ok, modify my copy – no weird ‘mutable’ needed
   …
}

int count = 0;
f( count );                                // ok, pass a copy
```

So there are three problems:

- Usability: It is surprising to users that the normal use of a captured variable is not allowed, and that *const* is injected without being mentioned anywhere.
- Clarity: The current syntax is ugly, and invents a new and different use of *mutable* (also a consistency issue).
- Consistency: The current syntax is inconsistent with the rest of the language, because nowhere do we disallow modifying an explicitly copied value.

Here is a possible resolution, which requires a breaking change.

(Good) Option 1: Do not make captured copies implicitly *const*. (And, since there is then no longer a need for the baroque *mutable*, remove that too.) The above examples would be allowed as written.

Breaking change impact assessment:

- All valid code that captures by value and does not use *mutable* continue to compile, but may select non-const overloads.
- All valid code that captures by value and does use mutable would only need to remove the now-extraneous *mutable*.