Document number P0588R0

Date 2017-02-05

Authors Richard Smith <richard@metafoo.co.uk>

Audience Evolution

Simplifying implicit lambda capture

Introduction

During discussion of constexpr if (P0252) at Oulu, a particularly problematic case was discovered:

```
template<typename T> void f(T t) {
  auto lambda = [&](auto a) {
    if constexpr (Copyable<T>() && sizeof(a) == 32) {
      T u = t;
    } else {
      // ... do not use t ...
    }
  };
  // ...
}
```

Suppose we instantiate `f(unique_ptr<int>)`. In order to complete the type of `lambda`, the set of implicit captures of the lambda must be known. To produce that set, we substitute `T` = `unique_ptr<int>` into the body of the lambda. However, the constexpr if condition produced by that substitution is still dependent -- it depends on `a` -- so we substitute into both *arms* of the if statement and immediately produce an error because it cannot be copied. This happens even if the lambda is never called.

This can be worked around with a source change:

```
-    if constexpr (Copyable<T>() && sizeof(a) == 32) {
+    if constexpr (Copyable<T>())
+      if constexpr (sizeof(a) == 32) {
```

… but it raises the question of whether our complicated (odr-use-based) implicit capture rules are really worth the complexity they introduce into the language, and whether we could accept the above case with a different set of rules.

Status quo

When a local variable is referenced, an implementation notionally walks outwards from the point of reference to the point of declaration of the variable as follows:

1. If the outward walk reaches a point where a load (lvalue-to-rvalue conversion) is performed, and one of the possible values for the load is that occurrence of the name of the variable (or a subobject that was derived in a syntactically obvious way), and loading the variable is possible within a constant expression[1], and the containing full-expression is not in any way dependent on a generic lambda parameter, the reference to the variable is replaced by that constant value and the walk terminates.
2. If the outward walk encounters an unevaluated operand (`decltype`, `sizeof`, `noexcept`, some forms of `typeid`), the walk terminates.
3. If the outward walk encounters a local class or a lambda that cannot capture the variable (because it has no capture-default), the program is ill-formed.
4. If the outward walk encounters a lambda that can capture the variable, then that lambda does capture the variable.

Note that the rules for lambda capture and for use of local variables within local classes are identical. Rules 2-4 are purely syntactic, and can be applied independent of the actual values of any template arguments. Rule 1 depends on knowing which uses result in a load, which cannot be known in general in a dependent expression, which is why it is effectively disabled for expressions depending on a generic lambda parameter.

Example:

```
void g(int);
void h(const int&);
void f() {
  const int n = 5;
  [](bool b) {
    g(b ? n : 1); // ok by rule 1
    int arr[n]; // ok by rule 2
    decltype(n) m; // ok by rule 3
    h(n); // ill-formed by rule 4, n is not loaded here
  }(true);
  struct S {
    void operator()(bool b) {
      g(b ? n : 1); // ok by rule 1
      int arr[n]; // ok by rule 2
      decltype(n) m; // ok by rule 3
      h(n); // ill-formed by rule 4, n is not loaded here
```

[1] In standardese, we say that the reference does not constitute an odr-use in this case.

```
    }
  };
}
```

Problem

Rule 1 does not naturally generalize to contexts that depend on template parameters. Prior to the advent of generic lambdas, this did not matter, because determination of whether Rule 1 applied could always be deferred until all template parameters were known. But generic lambdas create a unique situation: we must semantically analyze the body of the implied function template without knowing the parameters that will be used to instantiate it; that knowledge leaks out of the generic lambda by way of its captures.

Beyond that, determining the captures even for a non-dependent portion of a generic lambda requires implementation heroics from implementations that perform token soup instantiation -- they must somehow analyze the token stream of the lambda body to determine its capture set, which in general can require the full complexity of C++ declaration and expression analysis -- requiring something akin to careful archetype instantiation.

The complexity of these rules result in widespread implementation divergence.

Can’t computation of lambda captures be deferred until we know the template arguments of operator()?

Not in general. Consider:

```
void might_use(int, int);
void might_use(const int&, bool);
inline auto f(int n) {
  return [=](auto a) { might_use(n, a); };
}
```

This function might appear in multiple translation units in the same program, and in order for the program to work correctly, those translation units must all agree on the layout of the returned type and thus on its captures. But one translation unit might contain `f(0)(0)` where another contains `f(0)(true)`.

Possible resolution

The above problems could be resolved by removing Rule 1’s ability to block Rule 4, but retaining its ability to block Rule 3. Put another way, a lambda-expression would (formally) implicitly capture a variable from an enclosing scope when possible, *even if* it only uses that variable in contexts where it is immediately loaded and its value is usable in a constant

expression. If the variable cannot be captured, the program would still only be ill-formed if the address (and not just the value) is needed, as is the case today.

The proposed rule for implicit lambda capture of variables can be summarized as:

> When a variable is referenced outside of an unevaluated operand, if every lambda-expression between its declaration (possibly as an explicit capture of an enclosing lambda) and its use has a capture-default, then the variable is implicitly captured.

How does this solve the original problem?

With this change in place, the set of captures of a lambda can be determined by a walk of the syntactic structure of the lambda-expression[2], which means that we do not need to instantiate the body of the *lambda-expression* within `f(unique_ptr<int>)` in order to determine its capture set. Instead, we can defer substituting `T` = `unique_ptr<int>` into the body of the lambda until we also know the type of the generic lambda parameter, at which point we can skip the unchosen arm of the `if constexpr`.
We would essentially treat the body of a lambda as a separately-instantiated entity, as we do member functions of classes and the like, and instantiate it as needed instead of instantiating it eagerly with the enclosing context. Note that we propose applying this treatment uniformly to both generic and non-generic lambdas (this is not essential to the proposal, but seems to give a more consistent language rule).

Effect on existing code

This may be a breaking change for certain code patterns, as it may result in lambdas (formally) capturing variables that they did not previously need to capture. Three factors mitigate this:

- This only affects variables that can be used in a constant expression -- namely, const-qualified integer variables and constexpr variables. Therefore we know the type of the capture will be trivially-destructible and very likely trivially-copyable, so the additional copy performed by the capture is likely non-observable.
- Uses of the variable that satisfy Rule 1 are not transformed into uses of the capture, and so the semantics of the body of the lambda are unaffected.
- An implementation is permitted to optimize away unused lambda captures (see below).

The change would be detectable by code such as:

```
struct A {
  constexpr A(int n) : n(n) {}
  A(const A&) = delete;
  int n;
```

[2] This is still not trivial for token soup instantiation, as the variable name could be shadowed by a local variable. `typeid`'s "sometimes evaluated" nature also presents a unique problem that we might want to address -- perhaps simply by considering a dependent `typeid` to always be potentially evaluated for the purpose of lambda capture.

```
};
void f() {
  constexpr A a(0);
  [=] { int n = a.n; };
}
```

This was previously valid, but under the new rule, the lambda within `f` captures `a`, resulting in a call to `A`'s deleted copy constructor. However, both MSVC and Clang already reject the above program today, because they try to capture `a`.

Effect on implementations

Implementations can choose to compute the capture set for a lambda earlier if they wish (within a template definition, without regard to any particular instantiation), or defer the decision until the immediate context is non-dependent (as they are required to do today).

Under the as-if rule and [expr.prim.lambda]/4.1, an implementation is permitted to remove unused captures from a closure type. Therefore an implementation that wishes to do so can avoid capturing in most cases where they can avoid capturing today. The primary exception would be when knowing the value of an enclosing template parameter is necessary to determine the captures of the lambda, in which case the correct decision may be difficult or impossible to make without risking user-visible effects; another exception would be when copying the captured variable might introduce visible semantic effects.

One other impact should be called out: this change would introduce the first case in which we permit deferred instantiation of a portion of a function. In all other cases, entities within a function are either instantiated immediately with the function (this applies to local classes and their members, local enums, and so on) or not instantiated at all (this applies to the unchosen arm of a constexpr if). There are novel challenges here: unlike deferred instantiation of class member definitions, implementations may not have (in general) a mechanism to map from a usage of a variable in a local lambda to the corresponding variable in the instantiation of the enclosing function. **This is the key problem that makes local templates difficult**, and resolving it would require significant implementation complexity for at least some implementations. However, if we can make this work for local lambdas (and the difficulties appear to be implementation difficulty, not a fundamental issue with the idea), it would also remove the barrier to permitting local templates in general.

Acknowledgements

Thanks to Daveed Vandevoorde for reviewing an earlier version of this paper. Thanks (I suppose...) to EWG for requesting that Daveed and I investigate solving this problem.