Document number: P0927R1

Date: 2018-05-16

Reply-to: James Dennett <jdennett@google.com>
Geoff Romer <gromer@google.com>

Audience: Evolution Working Group

Towards A (Lazy) Forwarding Mechanism for C++

Synopsis

By allowing function declarations to specify that certain arguments are to be passed lazily, we can reduce the need for reliance on the preprocessor and simplify and improve generic code that forwards arguments. We anticipate that there will be other use cases.

While this is not a proposal for a terse lambda syntax, it does allow for the creation of callable objects without syntactic overhead.

By declaring a function parameter as being lazy, call expressions invoking that function will implicitly create callable objects. Within the called function these behave as any other callable. For example, evaluating them requires explicit use of the `()` operator.

Note that this is a discussion paper to solicit EWG direction, and does not propose wording.

Possible Syntax

Alternative syntaxes are possible. We pick one here to use in this discussion. The only crucial syntactic aspect is that lazy arguments have no syntactic overhead at the call site.

Declaration:

```
int log(bool, [] -> std::string message);
```

Call:

```
log(value > threshold,
    std::to_string(value) + " exceeds " + std::to_string(threshold));
```

Definition:

```
int log(bool condition, [] -> std::string message) {
  if (condition) std::cerr << message() << std::endl;
}
```

At the call site, we have the shortest possible syntax: the callable is denoted by the expression or braced-init-list (i.e., the *initializer-clause*). Where prior lambda proposals have hit difficulties in resolving the tension between wanting to preserve the value category of the expression (per `decltype(auto)`) versus the return type deduction rules for lambdas today (`auto` deduction, not `decltype(auto)`), the current proposal has a syntactic slot for the return type in the parameter declaration, allowing authors to specify whether the implicitly-generated callable will return by value or by reference.

Because type information is present in the declaration of the parameter, this can handle braced-init-list arguments as well as expression arguments: the context provides the required type for the braced-init-list.

When selecting viable functions and performing overload resolution, and for purposes of template argument deduction, the return type of the callable is used as the parameter type. Overloads on `T` and `[] -> T` are ambiguous (i.e., neither lazy nor eager parameters are preferred over the other).

Motivating Use Cases

Conditional evaluation

Many widely-used C++ libraries expose “functions” that conditionally evaluate one or more of their arguments. For example:

- C++’s `assert` macro (inherited from C) evaluates the given predicate only in debug mode.
- LLVM’s `DEBUG` executes the given code only if a certain flag is set.
- Boost’s `BOOST_TRIVIAL_LOG` evaluates its streaming input only if the specified logging mode is enabled.
- GoogleTest’s assertions evaluate their streaming inputs only when the assertion fails.

Our codebase has plenty of other examples, and yours probably does too. Of course, these “functions” are actually all macros; they cannot be ordinary C++ functions because all function arguments are guaranteed to be evaluated before the function is entered.

We will not reiterate here all the reasons preprocessor macros are undesirable, and just note that if we want to enable modern C++ code to entirely avoid macros, we will need to find a way to address these use cases.

Better overloading for short-circuiting and sequencing operators

C++ has long allowed overloading for most operators, but guidance has been to avoid overloading the short-circuiting operators `&&` and `||` and the sequencing operator `,` because user-defined operator overloads obey function call semantics, which do not (up to C++17) permit control of short-circuiting or of order of evaluation. Lazy parameters would remove this special case, making operator overloading more regular and allowing user-defined types to behave more closely to built-in types.

```
// (Eliding details of reference handling.)
MyType operator,(MyType lhs, []->MyType rhs) {
  return rhs();  // Evaluated now, while lhs was evaluated by the caller.
}
```

More-perfect-than-perfect forwarding of initialization

It is well-known that “perfect” forwarding does not handle braced initializer lists:

```
std::vector<std::vector<int>> v;
v.emplace_back({ 1, 2, 3 });              // Error
auto ptr = std::make_unique<std::vector>({1, 2, 3});  // Error
```

“Perfect” forwarding also prevents copy elision, and doesn’t handle a variety of more esoteric use cases, such as `NULL`, bitfields, and overloaded functions. Our proposal does not solve these problems in general, but does provide an alternative approach without those shortcomings for the common case where we wish to perfectly-forward an initializer.

For example, we can define a better form of emplacement:

```
template <typename T>
auto vector<T>::better_emplace_back([] -> T value) {
  __grow_if_needed();
  new (&__m_array[__m_size]) T(value());  // Copy elision is guaranteed here
  ++__m_size;
}

std::vector<std::vector<int>> v1;
v1.better_emplace_back({ 1, 2, 3 });  // Braced initialization works

struct S {
  unsigned int i : 5;
} s{0};
std::vector<int> v2;
```

```
v2.better_emplace_back(s.i);  // Bitfields work
```

We can also define a better `unique_ptr` factory function:

```
template <typename T>
std::unique_ptr<T> create_unique([] -> T in) {
  return std::unique_ptr<T>(new T(in()));
}

// The basic syntax is similar to make_unique, but the type is part of the
// argument instead of being a template parameter.
auto p1 = create_unique(std::string{"foo"});

// Braced initializer lists work here too.
auto p2 = create_unique(std::vector<int>{1, 2, 3});

std::vector<int> make_vector();

// Unlike with make_unique, the type can be deduced when appropriate.
auto p3 = create_unique(make_vector());

struct S {
  s() : i(0) {}
  s(s&&) = delete;

  unsigned int i : 5;
};

// Copy elision is guaranteed.
auto p4 = create_unique(S());

// Bitfields work.
auto p5 = create_unique(p4->i);

// A heap-allocated non-movable lambda. This isn't possible in C++17 without
// explicit use of `new`.
auto p6 = create_unique([s = S()] { ... });
```

The validity of these examples follows naturally from guaranteed copy elision, and the intuitive model that this is syntactic sugar for a lambda at the callsite:

```
auto p1 = create_unique([&] () -> std::string { return std::string{"foo"}; });
auto p2 = create_unique(
    [&] () -> std::vector<int> { return std::vector<int>{1, 2, 3}; });
auto p3 = create_unique([&] () -> std::vector<int> { return make_vector(); });
auto p4 = create_unique([&] () -> S { return S(); });
```

```
auto p5 = create_unique([&] () -> int { return p4->i; });
auto p6 = create_unique([&] () -> auto { return [s = S()] { … }; });
```

A non-motivating use case: terse callbacks

In principle, lazy parameters could also be used to pass some kinds of callbacks without the syntactic overhead of a lambda, e.g. `m_mutex.while_locked(m_x = 3, m_y = 2);`. We consider this poor API design: because lazy arguments are syntactically indistinguishable from ordinary arguments at the callsite, laziness should be used only as an optimization, and/or to enable more-perfect initialization, not as an essential part of the function's semantics. Laziness may have some user-visible semantic effect, but that should not be the primary purpose.

These are subjective judgements, so it doesn't appear to be possible to forbid these sorts of usages at the language level. However, we note that several issues will tend to discourage this sort of API design:

- Lazy arguments must be *initializer-clause*s (roughly: either expressions or braced-init-lists), so a lazy parameter will be much less flexible than a general lambda.
- Lazy arguments must be capable of initializing a parameter of some complete type, and so void expressions cannot be used as lazy arguments. The above example is viable only because `operator=` happens to have a non-void value, which in this context is more or less an accident; a call to a void function would not compile.
- APIs such as the above cannot easily be overloaded to take either an explicit lambda or a lazily-evaluated argument, because the two overloads will typically be considered ambiguous (although this can be worked around with SFINAE).

Callsite Syntax

Why not solve this with "terse lambdas"?

We anticipate objections to this proposal on the grounds that we should instead focus on providing a lambda notation that's terse enough that programmers will be willing to use it at the callsite. We do not believe this is a compelling direction for the use cases we are interested in; addressing them in a way that is not coupled to the function declaration would require major changes to the semantics of lambdas.

The key point to realize is that the arguments to a function are not *expressions*, they are *initializers*, and the semantics of an initializer very often depend on the type being initialized (braced initializer lists are the most obvious example, but not the only one). Thus, many of our use cases cannot rely on return type deduction; either the return type must be specified explicitly at the callsite (which users will flatly refuse to do, even leaving aside the cases where naming the type is impossible or absurdly difficult), or it must be obtained from some other source. Our proposal hinges on the fact that the return type of the callable is specified by the function signature, so the user need not specify it, and deduction is not necessary.

The only way we see to address our use cases through extensions to lambdas would be to extend lambdas with a terse syntax for treating the lambda's return type as a non-deduced template parameter, so that the return type can be specified when the lambda is invoked, rather than when it is created. Absent additional core language changes, this would require the caller to invoke the lambda function body using the more awkward `operator()` syntax, because an overloaded `()` operator cannot take an explicit template parameter when invoked by ordinary function call syntax.

Note that these changes would be *in addition to* the sort of terseness changes discussed in other proposals (at a bare minimum, we believe such a syntax would need to permit the user to omit the `return` keyword if it is to have any meaningful uptake).

Furthermore, by making the lambda explicit at the callsite, we would make it much harder to avoid the compilation overhead of synthesizing a full lambda class type for every use of a lazy parameter. This overhead could be quite costly, because many of the constructs in our motivating use cases are used very frequently. By hiding the lambda from the caller (and perhaps also from the callee), we make it much easier to facilitate (or even mandate) optimizations that avoid that overhead.

By the same token, absent significant changes, any solution in which the arguments are passed "by lambda" would require the called function to become a template. We do not propose to require that; a function taking a lazy parameter is still just a regular function. Compilers may be able to generate specializations at call-sites if the function body is accessible, e.g., if it is inline.

Compared to our proposal, this lambda-based approach would be substantially more awkward and invasive, probably more costly to compile, and require us to take a dependency on a hypothetical feature (terse lambdas) whose future is uncertain at best, and offers no offsetting benefit. Consequently, we do not think this is a good direction to pursue. Our position might change if there were other important use cases for the feature discussed above (generic lambdas with caller-specified return types), but so far we are not aware of any.

Should the callsite be marked somehow?

Lazy parameters do pose some unique hazards for the user. In particular, if the evaluation of an argument has side effects, those side effects may not happen, or may happen in a different order, depending on the semantics of the function. For example, users occasionally run into trouble with `assert()`-style macros because they unwittingly rely on side effects of evaluating the predicate, so everything works until they build in opt mode.

One possible response to this would be to require lazy evaluation to be somehow marked at the callsite, so that the user has to explicitly opt into these changes to argument-passing semantics. However, we don't think that's appropriate, for several reasons:

- Functions that unconditionally evaluate all their arguments (such as the perfect forwarding use cases discussed elsewhere) pose very little risk; user code could in principle observe the differences in how the evaluation is sequenced, but we expect such code to be very rare. Requiring such low-risk usages to be marked up will tend to degrade the value of that markup, because they are effectively false positives.
- Mutable reference parameters are vastly more hazardous to an unwary programmer than lazy parameters, and yet reference parameters do not require callsite markup, and experience has shown that this was probably the right decision; C++ API designers have generally used mutable reference parameters judiciously, using function naming and documentation to minimize the risk of misunderstanding or misuse. We expect the same to be true of lazy parameters.
- Programmers might see the callsite markup as unwelcome boilerplate, and prefer to stick with macro-based solutions.
- Those who want to require callsite markup in their code can do so without the standard's help, by implementing a compiler/linter warning that checks for a particular attribute. Such an approach could be much more pragmatic than a blanket mandate in the core language; for example, perhaps individual parameters could opt into (or out of) this warning depending on how risky they are judged to be.

Interactions with the type system

Is laziness part of the parameter's type?

From the caller's point of view, a lazy `T` parameter should be indistinguishable from an ordinary `T` parameter, at least with respect to name lookup, overload resolution, and template argument deduction. One may therefore be tempted to treat laziness as separate from the type system, perhaps something more like a calling convention. However, this raises some major problems; for example, it would be exceedingly difficult to permit a `void(*)(T)` to point to a function that took a `T` parameter lazily. By much the same token, it is likely that users will need to introspect on the laziness of a function's parameters for metaprogramming purposes. We therefore think that laziness should be part of the type system, i.e. a lazy parameter's type should be distinct from the type of any non-lazy parameter.

Must a lazy parameter have a single type?

The most natural mental model for lazy parameters in terms of existing C++ concepts is to think of each lazy argument as being reified as a lambda, i.e. `f(arg_initializer)` becomes `f([&] { return arg_initializer;})`. That naturally suggests that for a given lazy parameter, different arguments will have different types (and so a function with a lazy parameter must at least implicitly be a template). However, as we alluded to earlier, requiring the compiler to synthesize a full lambda type for each argument could be very costly for widely-used functions, and the requirement that the function be a template could be onerous for some use cases.

Furthermore, it appears to be unnecessary; making each argument a distinct type does not improve type-safety, and there is no legitimate need for e.g. the destructor (which is trivial) or the move constructor (which is useless, since the argument cannot safely outlive the expression in which it was created). Any lazily-evaluated `Foo` argument can be straightforwardly represented with two pointers, a `Foo(*)(void*)` representing the code to execute and a `void*` to pass to it, representing the captured references, so it's sufficient to have a single lazy counterpart for each non-lazy type (this representation is strikingly similar to the expected representation of P0792's `function_ref`). Admittedly, making each argument a distinct type could facilitate some optimizations, by effectively forcing inlining of every call, but the same effect can be achieved in a more flexible way by marking the function `inline` (possibly coupled with an always-inline attribute, if the compiler would otherwise make the "wrong" inlining decision).

Consequently, we think a lazy parameter should have a single type for all arguments, at least in the common case.

Is a lazy parameter type a class type?

It seems natural to treat `[] -> X` as a kind of class type, since it is callable but not a function or function pointer. However, at least one vendor argues that standard class types should be specified and implemented in the library rather than the core language (lambdas being an unavoidable exception), so if we want it to be a class type, we may wish to spell it more like `std::lazy<T>`. Such a library type would be comparable to `std::initializer_list` in that it has special core-language rules for initialization. However, it would go beyond that precedent by also having special rules for name lookup, overload resolution, and template parameter deduction.

Alternatively, we could treat `[] -> X` as a new kind of object type, whose only operation is postfix `()`.

Can lazy `[] -> X` types be used outside of parameter lists?

The type itself certainly can, but we do not propose at this time that this *syntax* be valid in other contexts. We do aim not to preclude it. In particular, it is desirable that any syntax selected does not depend on the restriction to parameter types.

Should invocation be &&-qualified?

It is usually not safe to invoke a lazy parameter more than once, because the argument's *initializer-clause* may move from one or more local variables when invoked. For example:

```
void fn([] -> std::unique_ptr<int> i);
…
std::unique_ptr<int> ptr = …;
fn(std::move(ptr));
```

If the body of `fn` invokes `i` twice, the second invocation will return null. As one of us has argued in N4159, we generally believe that function objects that cannot safely be invoked multiple times should generally be callable only as rvalues, in order to make this requirement explicit in the type system, and improve safety by making any subsequent call a use-after-move error.

However, in this case the call-once requirement is already present in the type system, because it's implicit in the fact that the parameter is lazy. Requiring an explicit `std::move` at the callsite might help programmers avoid mistakes (and make mistakes more obvious), but this requirement could easily be seen as "boilerplate", since it would be required on every use of every lazy parameter. Furthermore, there is currently no way to define a lambda that is callable only as an rvalue, so if the function objects produced by this feature had that property, it would erode the intuitive equivalence between this syntax and an explicit lambda at the callsite.

Consequently, we tentatively recommend permitting lazy parameters to be callable as lvalues.

Should lazy types be copyable/movable?

Copying or moving of lazy types would necessarily be "shallow", since we cannot in general copy the local state that the argument initializer-clause refers to. This makes copying especially problematic, since the implicit requirement that a lazy parameter be invoked at most once would now apply across all copies of that parameter. Moving appears to be more benign, but still troublesome: moving a lazy parameter could all too easily enable it to outlive the state it refers to. By the same token, it seems very likely that any safe use case for lazy parameters could be implemented without copying or moving them, but instead copying/moving pointers to them when necessary (this would be especially true if we permitted lazy parameter pointers to be callable, like function pointers).

In the absence of a compelling use case, we recommend making lazy types non-copyable and non-movable. (Note: it is possible that further consideration of generic code might change this position in favor of allowing copies or moves.)

Is there a feature test macro for this?

This is postponed until the proposal is more mature.

Revision History

R1: Added motivating example of overloading short-circuiting and/or sequencing operators.

Acknowledgements

We thank Richard Smith and Matt Calabrese for their feedback on drafts of this document.