Proposal for Generic (Polymorphic) Lambda Expressions
Document no: N3418=12-0108

Faisal Vali Herb Sutter Dave Abrahams

2012-09-21

Abstract: The C++11 standard provides support for non-generic lambda expressions. In the spirit of the original proposal, and motivated by the design principles that led to the introduction of `auto`, we propose adding generic lambdas to C++.

1 Introduction

C++11 lambda expressions can considerably improve the syntax of code that uses simple, one-off, function objects. Unfortunately, using lambdas can be frustrating because the programmer is often required to explicitly spell out parameter types that the compiler could deduce automatically. Here follow two examples:

```
std::for_each( begin(v), end(v), [](decltype(*begin(v)) x){ std::cout << x; }
);

auto get_size = [](
  std::unordered_multimap<std::wstring,std::list<std::string> > const& m
 ){ return m.size(); };
```

If the lambda expressions were generic, these examples would become much simpler:

```
std::for_each( begin(v), end(v), [](& x){ std::cout << x; } );

auto get_size = [](& m){ return m.size(); };
```

The lack of genericity in C++11 lambdas can also be a direct obstacle. For example, Consider a `std::transform`-like algorithm whose implementation has been vectorized with SIMD instructions. The length of most input sequences will not be an exact multiple of the SIMD vector size, so two versions of the function parameter are needed: one for complete vectors and another for scalar elements. To accommodate C++11 lambdas, the algorithm might accept two functions:

```
vtransform(
  begin, end,
  [](float x){ return 2*x; }, [](simd_vector<float> const& x){ return 2*x; }
);
```

Note that in a correct call to this algorithm, the bodies of the two functions will always be identical. With a generic lambda, both functions could be supplied by a single argument:

```
vtransform( begin, end, [](&x){ return 2*x; } );
```

Generic lambdas have precedent in other useful statically-typed languages:

C# 3.0 (2007)	:	`x => x * x;`
Java 1.8 (~2013)	:	`x -> x * x;`
D 2.0 (~2009)	:	`(x) { return x * x; };`

In addition, members of the C++ committee and broader community have expressed the need for generic lambdas across various C++ forums.*

Generic lambdas similar to those proposed here were considered in the original C++11 lambda proposal [Willcock2006], but were eventually dropped for C++11 because it wasn't clear how an un-constrained lambda expression could be used inside a concept-constrained template [Toronto 2007, Järvi2007]. Although concepts were *also* removed from C++11, the community still has an interest in seeing concepts in a future C++, and the committee is still working on the problem [Stroustrup 2012], so conflicts between these two features can't be dismissed out-of-hand. Fortunately, we believe we now know how to type-check uses of un-constrained templates from constrained ones [Järvi2011-19566, Abrahams2011-20760, Abrahams2011-next].

2 Proposal

We are proposing the following pure extensions to C++11:

1. Allow the *type-specifier* to be omitted in a lambda's parameter declaration (as in the examples above), thereby making the lambda generic
2. Allow the use of a function template parameter list and familiar template syntax, making it possible to for generic lambdas to be partially-ordered
3. Allow the lambda body to be an expression
4. Allow regular function templates to be declared with a lambda-like syntax

Details follow.

2.1 Allow the type-specifier within a parameter declaration of a lambda to be optional (as intended by the original authors of C++11 Lambdas[1])

If the initial *type-specifier* within the *decl-specifier-seq* of a lambda's *parameter-declaration* is omitted, the expression creates a generic closure. A generic closure type is just like a familiar C++11 closure type except that its function call operator is a member function template. In the operator's declaration, omitted *type-specifier*s are replaced by unique template type parameters, which are added to the operator's *template-parameter-list*.

For example, this expression

```
[](const& x, & y){ return x + y; }
```

[1] Lawrence Crowl: "Yes. The intent was to expand lambda in just this direction after 2012."

might create the following closure type:

```
struct anonymous
{
    template <typename T, typename U>
    auto operator()(T const& x, U& y) const -> decltype(x+y)
    { return x + y; }
};
```

Note: the syntactic rule for what can be omitted from a lambda's *parameter-declaration* corresponds to the rule for the placement of `auto` in variable declarations: it must be the *type-specifier* at the beginning.

As a consequence of this rule, the following examples are all valid generic lambda expressions:

```
[](/*T*/ &&a                          ){}
[](/*T*/ *a                           ){}
[](/*T*/ const* a                     ){}
[](/*T*/ *const a                ){}
[](/*T*/ (&a)[5]                 ){}
[](/*T*/ (*funptr)(int, char)    ){}
[](/*T*/ X::*data_member         ){}
[](/*T*/ (X::*member_function)(int)){}
```

While these forms of *parameter-declaration* are expected to be less common than *x*, *&y*, and *const&z*, they can be partially-ordered. We explain why that may be useful in the next section. Note: This syntax intentionally does not support unnamed parameters.

2.2 Allow the use of familiar template syntax in lambda expressions

While omitting *type-specifier*s is very convenient, and expressive enough for most cases, it does not support certain very common parameter forms (`e.g shared_ptr<T>`, `initializer_list<T>`) that are useful in creating partial orderings amongst overloaded generic functions. Lambdas, being functions and not function objects, can't be overloaded in the usual implicit way, but they can be "explicitly overloaded" using the following simple code (brought to our attention by Mathias Gaunard):

```
template<class F1, class F2> struct overloaded : F1, F2
{
  overloaded(F1 x1, F2 x2) : F1(x1), F2(x2) {}
  using F1::operator();
  using F2::operator();
};

template<class F1, class F2>
overloaded<F1, F2> overload(F1 f1, F2 f2)
{ return overloaded<F1, F2>(f1, f2); }
```

This technique would especially useful for creating visitors, e.g. for boost::variant, something like this:

```
auto visitor
  = overload( [](&a)[N]) {...},
    overload( [](T* p) { ... },
    overload( [](shared_ptr<T> sp) { ... },
       [](initializer_list<T> il) { ... })
    ));
```

However, simply omitting template parameters doesn't allow us to express the lambda expressions needed for this example (note the uses of "*T*" and "*N*"). Therefore, we propose to allow a *template-parameter-list* enclosed in angle-brackets following the *lambda-introducer*, as follows:

```
auto visitor
  = overload( []<int N>((&a)[N]) { … },
    overload( []<class T>(T* p) { … },         // equivalent to [](*p){ … };
    overload( []<class T>(shared_ptr<T> sp){ … },
              []<class T>(initializer_list<T> il){ … }
    )));

visitor( {1, 2, 3} ); // ok - calls initializer_list "overload"
```

When mixed with explicit *template-parameter-list*s, template parameters implied by omitted *type-specifier*s are appended to the list, so `[]<int N>((&a)[N])` is equivalent to `[]<int N, class T>(T (&a)[N])`.

Familiar template syntax and partial ordering rules apply, with no new rules or corner cases, so we expect all of the following examples to be valid:

```
[]<int N>(int (&a)[N]) {}

[]<class T, int N>(T (&a)[N], b) {}

[](Stack<std::vector, int, 5> x, a) {}

[]<int N, class T>(Stack<std::vector, T, N>& a, b) {}

[]<
    template< template<...> class, class, int> class StackLike
    , template<...> class V, class T, int N
  >( StackLike<V, T, N>& a, b) {}

[]<class...Tys>(Tys...a) {}
```

Since the function call operator is public, elements of the *template-parameter-list* can be explicitly specified if needed:

```
[]<int N>(const int(&h)[N]){}).operator()<5>({1,2,3});)
```

In early 2009, a similar experimental extension to lambda expressions was implemented for GCC (http://gcc.gnu.org/ml/gcc/2009-08/msg00174.html).

2.3 Permit a lambda body to be an expression

Experience has shown that many lambda bodies are short, and of the form "`{ return` *`expr`*`; }`". As a further convenience, we propose allowing such a lambda body to be written as simply "*expr*" with identical semantics:

```
for_each( begin(v), end(v), [](&e) e += 42 );      // { return e += 42; }

sort( begin(myints), end(myints), [](i,j) j < i ); // { return j < i; }
```

Note: this extension should also apply to non-generic lambdas and should not be tied to generic-lambdas.

2.4 Allow regular function templates to be defined by inserting a name in the (extended) lambda syntax

The great syntactic convenience of the extensions proposed in here, and of the existing automatic return type deduction, should be extended to ordinary function templates, allowing declarations like this one:

```
[]min(&x, &y) y < x ? y : x;      // define a function template
```

In this syntax, non-empty capture lists would be forbidden. See also http://cpp-next.com/archive/2011/11/having-it-all-pythy-syntax/, where this extension was first proposed, and http://pfultz2.github.com/Pythy/, where it has been implemented in terms of macros.

2.5 Autogenerate a SFINAE-friendly template conversion operator to pointer-to-function in captureless generic lambdas

Similar to the C++11 non-generic lambda, each polymorphic lambda that does not capture any variable shall have a public non-virtual non-explicit const conversion to pointer to function, that, when invoked, has the same effect as invoking the closure type's function call operator. In the case of a generic lambda, this conversion function will be a member template that provides a SFINAE-friendly conversion to a type-appropriate function-pointer using current C++11 rules. For example,

```
void (*fp)(widget, widget*, gadget const&, gadget (*)(widget&)
         , widget (&)[10], int (widget::*)()
          ) = [](a, b, c, d, e, f) { };
```

3 Member templates of local classes

We define the semantics of the proposed generic lambdas as similar to that of non-generic lambdas except that the function call operator and implicit conversion to function-pointer (if capture-less) will be generated as member templates.

Since block-scoped lambdas are defined in terms of local classes, we'd need to ensure that the semantics of member templates of local classes (which are not supported by C++11) are well defined and consistent with those of member templates of non-local classes. While this issue has presumably been solved, since member templates of local classes seem to be supported by Clang-3.2, we have largely sidestepped this issue by relying on our current C++11 intuition of the behavior of member templates. We assume that if this proposal is moved forward by EWG, these details will need to be fully-elaborated (probably in a separate proposal) before this feature can be incorporated in the working paper.

There have been some suggestions on the reflectors [Voutilainen2011-20750, Denett2011-19528, Gabriel2011-19533] of extending the language to support local templates. If such a proposal is made and accepted, then this particular issue would evaporate.

4 Implementation issues

Although this is a preliminary design paper, we spent some time considering implementation. Naturally, if this paper is moved forward by EWG, more attention would be given to this area, but so far, we believe the following implementation issues might need further discussion and investigation:

1. The grammar rules for lambdas with omitted *type-specifiers* and lambda bodies as expressions would need to be explored for feasibility and ambiguity (although mention has been made in this web-article that some of these rules have already been worked out)
2. The feasibility of return type deduction of generic lambda-expressions would need to be investigated
3. Notwithstanding the patch mentioned earlier, there is currently no complete implementation of these features in a C++ compiler. Some of the authors of this paper are working on a C++ compiler implementation and intend to share their experience in a followup paper.

5 Design Space Considerations, Choices and Rationale

In this section we discuss some of the design choices we considered and our rationale for their omission where applicable.

5.1 We do not require or allow the use of a placeholder such as `auto`

We considered the use of `auto` as a type specifier to indicate a generic parameter and decided against it because we felt that it did not add readability.

Consider the following proposed forms:

```
[](a)                       vs    [](auto a)
[](&a)                      vs    [](auto &a)
[](&&a)                     vs    [](auto &&a)
[](*a)                      vs    [](auto *a)

[]((&a)[5])                 vs    [](auto (&a)[5])
[]((*funptr)(int, char))    vs    [](auto (*funptr)(int, char))
[](X::*memvar)              vs    [](auto X::*memvar)
[]((X::*memfun)(int))       vs    [](auto (X::*memfun)(int))
```

A consequence of this decision is that generic lambdas do not support unnamed parameters.

It might be that some would prefer to have *auto* be optional here. If that is the case, we would remind them to consider the discrepancy in deduction semantics between *auto variable deduction* and *template argument deduction* for function calls when it comes to initializer_lists, and how the *auto* might seem misleading if generics follow *template argument deduction* for function calls. This would be a non-issue if there was a perfect correspondence between *auto deduction* and *template argument deduction* for function calls, and we hope this discordance will be addressed by another proposal.

In addition, if after feedback from the core-language committee, it is felt that the use of *auto* is essential to avoid parsing ambiguities, then we would obviously need to reconsider the implications of this design choice.

5.2 We do not propose allowing instantiation of a non-generic lambda from a generic one

We considered providing a member function template, which for the purposes of this section we will call *instantiate*, within the closure type of every generic lambda, that if supplied (either explicitly, or implicitly) with types for the generic parameters would instantiate a non-generic lambda closure object of a unique closure type.

That is, imagine a nested member template class within the closure type of the generic lambda, that has the same *template-parameter-list* as the inline function call operator and contains its own non-template inline function call operator with corresponding types that forwards to the generic lambda's body. Thus, this type when specialized with the relevant types, would instantiate a non-generic version of the generic lambda, and behave exactly like any other non-generic lambda.

So consider:

```
struct widget { ... } w1;  struct gadget { ... } g1;
struct widget2 { ... } w2; struct gadget2 { ... } g2;

auto generic = [](a, b) { ... }

generic(w1, g1); // ok
generic(w2, g2); // ok

auto nongeneric = generic.instantiate<widget, gadget>();

nongeneric(w1, g1); // ok
nongeneric(w2, g2); // ERROR
```

While the idea seemed vaguely promising to some of us, none of us could construct a compelling use-case, so we dropped it. Without this feature, if a non-generic version of the lambda is truly ever needed, perhaps a non-generic lambda expression could be made to forward to the generic lambda, e.g:

```
auto g = [&local_capture](a) { return a + local_capture; };
auto n = [g](int a) { return g(a); };
```

5.3 We note that generic lambdas support recursion slightly better than non-generic lambdas

Currently, in C++11, if one desires a recursive lambda expression, then one has to resort to the following gymnastics using std::function. e.g.,

```
std::function<int(int)> factorial;
factorial = [&factorial](int n) { return n <= 1 ? n : n * factorial(n -
1); };
int n = factorial(5);
```

This seems hackish and is less efficient than the direct use of the lambda.
With generic lambdas, such affairs improve slightly, and one could write:

```
auto factorial = [](self, n) { return n <= 1 ? n : n * self(self, n-1); }
int n = factorial(factorial, 5);
```

Not too much better, and in some ways worse as far as syntax goes, but it avoids the overhead of std::function and a capture.

5.4 Feature extension in the setting of the Concepts Proposal (n3351)

Since the initial generic-lambda proposal was affected by concerns regarding interactions with concepts, we felt that we should discuss the ramifications of the current Concept design on our proposal. While the implementation concerns are discussed in the section on implementation issues, we wanted to briefly touch upon how generic-lambdas could evolve in the setting of concepts.

For concepts with a single template parameter, one could write:

```
concept PodType<typename T> = is_pod<T>::value;

[](PodType p) { } // generic lambda accepting only pod types
```

For concepts with multiple template parameters one could resort to the more verbose syntax described above.

6 Further work

We are certain that we have not covered the entire extent of the design space here and welcome suggestions to refine and improve this proposal. In particular, we believe we may have made an important step towards the unification of functions and function objects in C++, but that work is still ongoing.

7 Acknowledgments

We are grateful for the timely help and comments provided by Dean Michael Berris, Lawrence Crowl, Mathias Gaunard, Jaako Järvi, Ville Voutilainen.

Arthur Butcher is the author of the GCC patch mentioned earlier in the paper.

This proposal draws much from all the initial lambda (generic and nongeneric) proposals put forth by Jaako Jarvi, John Freeman & Lawrence Crowl.

(*) User Demand

- "I really hope this gets added to C++ soon. This would fix so many problems with lambdas. Currently, lambdas in C++, trap you in a monomorphic box that you can't get out. So, in C++11, I have to use named functions and Boost.Phoenix, just like I did in C++03." Paul (http://cpp-next.com/archive/2011/11/having-it-all-pythy-syntax/)
- http://cpp-next.com/archive/2011/11/having-it-all-pythy-syntax/
- http://pfultz2.github.com/Pythy/ (Macro-based simulation of pythy functions)
- http://stackoverflow.com/questions/4643039/c11-and-the-lack-of-polymorphic-lambdas-why
- http://stackoverflow.com/questions/3575901/can-lambda-functions-be-templated
- "Why aren't there polymorphic lambdas?" (Scott Meyers' question every single time Herb shows any lambda example in a talk or paper)

References

- [Willcock2006] J. Willcock, J Järvi, D Gregor, B Stroustrup and A Lumsdaine. Lambda expressions and closures for C++ N1968=06-0038, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2006.
- [Järvi2007] J Järvi, J Freeman, and L Crowl. Lambda expressions and closures for C++ (Revision 1) N2329=07-00189, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2007.
- [Stroustrup2012] B. Stroustrup, A. Sutton (editors). A Concept Design for the STL N3351=12-0041, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2012.
- [Järvi2011-19566] c++std-core-19566 Re: Why [temp.mem] p.2: A local class shall not have member templates. ?
- [Abrahams2011-20760] c++std-core-20760 Re: Template definitions in block scope?
- [Abrahams2011-next] http://cpp-next.com/archive/2011/12/a-breakthrough-for-concepts/
- [Toronto2007] : C++ ISO Standards committee Evolution WG wiki, Thursday Notes Excerpt:
  debate about monomorphic/polymorphic lambdas and specification size and time.
  Jakko will redraft proposal.
  Two level strategy:
  1. simple syntax + just monomorphic design
  5/2/2/0/0
  2. general syntax + polymorphic
  Just make sure 1 is extendible to 2.
- [Voutilainen2011-20750]: c++std-core-20750, Template definitions in block scope?
- [Denett2011-19528]: c++std-core-19528, Re: Why [temp.mem] p.2: A local class shall not have member templates. ?
- [Gabriel2011-19533]: c++std-core-19533, Re: Why [temp.mem] p.2: A local class shall not have member templates. ?