Proposal for Assorted Extensions to Lambda Expressions
Document no: N3560

Faisal Vali Herb Sutter Dave Abrahams

2013-03-17

Abstract

We propose extensions to lambda expressions motivated by the general view that lambda expressions should allow for a concise and complete description of a callable unit of computation. In addition to containing those features from document N3418: Proposal for Generic (Polymorphic) Lambda Expressions that received *against* votes in Portland (2012), it also contains other small extensions to lambdas. We also present our experience implementing some of the features using Clang.

1 Introduction

The aim of this paper is to propose various extensions to lambda expressions (generic and non-generic) to simplify and enhance their use. This proposal builds on N3559 (which defines a generic lambda). We assume the reader is familiar with C++11 lambdas and the content of N3559.

It is important to note that not all the authors of this paper are in favor of each and every one of these proposed features and the purpose of this paper is to seek guidance from the EWG regarding which features are worth pursuing further.

2 Proposals

We are proposing the following orthogonal extensions to lambdas:

1 Allow the use of familiar template syntax in lambda expressions (implemented)
2 Permit a lambda body to be an expression (implemented)
3 Allow auto forms in the trailing return type
4 Allow generic lambdas with variadic auto parameters

Each of these is discussed in some detail in the subsections below.

2.1 Allow the use of familiar template syntax in lambda expressions

We propose to allow a *template-parameter-list* enclosed in angle-brackets following the *lambda-introducer*, as follows:

```
// LastElement demonstrates the use of a template-parameter-list.
// It accepts a std::array of any type and size, and returns its
// last element
auto LastElement = []<class T, size_t N>(const std::array<T,N>& a)
                  { return N ? a[N-1] : throw "index error"; };

int three = LastElement(std::array<int, 3>{{1, 2, 3}});
char B    = LastElement(std::array<char, 2>{{'A', 'B'}});
```

When mixed with explicit *template-parameter-list*s, template parameters implied by the use of *auto* as a *type-specifier* are appended to the list, so `[]<int N>(auto (&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 C++ programmers can rely on their intuition with templates to guide them in their use of this feature.

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});)
```

This additional syntax would have the following advantages :

1 — allow for certain type constraints to be placed on each parameter so that
errors can be detected at the point of call of the lambda and not at instantiation time

2 — allow for more sophisticated relations (such as *deduced* as same type)
between parameter types (vs the asymmetric case [](auto a, decltype(a) b)

where only the type of 'a' is deduced)

3 — allow for simplicity (by avoiding the quirks of **`decltype`**) in using the types of the generic parameters within the body of the lambda

4 — allow for partial ordering of generic lambda function call operator templates when overloaded within a class that inherits from multiple closure types

5 — can be extended naturally to support the concepts-lite proposal (without requiring a new *concepts* keyword)

6 — allow for forwarding lambdas (which require variadic packs)

A brief discussion and illustrative examples follow for each of the claimed advantages.

Advantage #1: Allow certain type constraints to be placed on each parameter so that errors can be detected at the point of call of the lambda and not at instantiation time of the lambda.

Consider this contrived example:

```
// A Lambda at namespace scope that returns another lambda
// that captures 'v' by reference and when passed a shared ptr
// adds it to the vector and returns the use_count of the
// shared_ptr.
//
// This code is merely for illustrative purposes.
// make_shared_ptr_pusher could be a function template
// assuming return type deduction per N3386

auto make_shared_ptr_pusher = []<class T>(std::vector<T> &v)
{
  return [&v]<class U>(shared_ptr<U> p) {
    v.push_back(p);
    return p.use_count();
  };
};

int main() {
  std::vector<shared_ptr<int>> v;
  auto pusher = make_shared_ptr_pusher(v);
  for ( int i = 0; i < 10; ++i )
    pusher(make_shared_ptr(i)); // ok at point of call

}
```

In the above example, *make_shared_ptr_pusher* creates a generic lambda that is constrained to accept only shared_ptrs; use of such a lambda with a non-shared_ptr will result in an error at the point of call and not at the time of instantiation of the lambda's body (thus allowing for better error messages). One can imagine various factory functions that create constrained generic lambdas that can be used at various different points in a program, and constraining them to accept only certain kinds of parameter types can be useful for detecting misuse early.

Advantage #2: Allow more sophisticated relations (such as deduced as same type) between parameter types.

As John Spicer [Spicer-ext-14260] stated on the EWG reflector (quoted with his permission): "*I think it is an important feature to be able to specify that multiple arguments of a lambda are required to have the same deduced type. Otherwise you will have cases that end up in the lambda with types that will result in errors instantiating the lambda rather than the more desirable point of call of the lambda*."

One could claim that relating parameter types to each other can be done with the use of **`decltype`** but the semantics of such a construct are different from the semantics of a generic lambda that uses the familiar template syntax (it is required to have the parameters maintain those relations during the deduction of the type). Consider this example:

```
auto Mult1 = []<class T>(T a, T b)        { return a * b; };
auto Mult2 = [](auto a, decltype(a) b)    { return a * b; };


int main() {
    struct X {
      operator int() const { return 0; }
    };

    X x;
    Mult1(3, 4); // ok
    Mult1(3, x); // Not ok.3 & x must deduce to same type (no
conversions)

    Mult2(3, x); // ok. Param 1 deduced as int, and 'x' is converted
}
```

As John Spicer points out (in the same thread): "*That is asymmetric in that it forces the second parameter to have the same type as the first. That is better than nothing, but I'm not sure it is good*."

Advantage #3: Allow for simplicity (by avoiding the quirks of decltype) in using the types of the generic parameters within the body of the lambda.

If one needs to name the type inside the lambda, it would be clearer to use the explicit template syntax than `decltype`. Consider this example:

```
auto MemFunPtr1 = [](auto& a) {
   // deal with the quirks of decltype
   using T = typename remove_reference<decltype(a)>::type;
   return &T::f;
};

// This is clearer and easier to specify
```

```
auto MemFunPtr2 = []<class T>(T& /*a*/) {
   return &T::f;
};
```

Advantage #4: Allow for partial ordering of generic lambda function call operator templates when overloaded within a class that inherits from multiple closure types.

While using **auto** as a *type-specifier* is quite convenient, it does not support certain very common parameter forms (e.g `shared_ptr<T>`, `initializer_list<T>`, `T (&)[N]`) that are useful in creating partial orderings amongst overloaded generic functions. Lambdas, being function objects and not functions, can not be overloaded in the usual implicit way, but they can be "explicitly overloaded" using the following simple code :

```
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 be especially useful for creating visitors, e.g. for `boost::variant`, which we expect to be a common use case for lambdas. Therefore, we propose to allow a *template-parameter-list* enclosed in angle-brackets following the *lambda-introducer*, as follows:

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

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

Advantage #5: Can be extended naturally to support the concepts-lite proposal (without requiring a new *concepts* keyword)

The explicit template syntax could naturally be extended to support the current work being done on constraints (concepts-lite) with Generic Lambdas; although, concerns have been raised about its verbosity. For e.g. the template syntax could be extended to support the following code (which assumes the reader is familiar with the aforementioned concepts-lite proposal):

```
auto find = []<Sequence S, class T> requires
                Equality_comparable<T, Value_Type<S>>()
                        (S&& seq, const T& v) { … };
```

Advantage #6: Allow for forwarding lambdas (which require variadic packs).

It is also worth noting that should generic lambdas be used to wrap either pre or post-actions around functions calls, forwarding arguments might be necessary, and the use of the explicit *template-parameter-list* would allow "perfect forwarding". Consider:

```
auto add_pre_printer =
        [](auto* objptr) {
            return [=](auto mem_fun) {
               return [=]<class ... Ts>(Ts&& ... args) {
                  variadic_print(args ... );
                  return (objptr->*mem_fun)(static_cast<Ts&&>(args)...);
               };
           };
        };

struct X {
   void f(int i, char&& c, X& x) { print("X::f\n");     }
   void g(X x)                   { print("X::g(x)\n");  }

  // used by variadic_print, if it can't find an appropriate
  // print specialization (i.e. print(int), print(char) etc)
  const char *to_string() const { return "X"; }
 };

 X x;

 auto X_add_mem_fun = add_pre_printer(&x);
 auto Xf = X_add_mem_fun(&X::f);
 auto Xg = X_add_mem_fun(&X::g);

 Xf(3, 'a', x);
 Xg(x);

 This prints (in our implementation):
  3, a, X
  X::f(3)
  X
  X::g(x)
```

We admit that supporting the *full template parameter list* feature has been deemed controversial (the Portland 2012 straw-poll outcomes were: 7 SF, 5 F, 3 N, 1 A, 1 SA[1]) by a few committee members, and therefore conclude this sub-section with some quotes from a committee member who was not present in the room during EWG's discussion of this feature in Portland.

Consider the recent words of John Spicer [std-ext-14263]: "*I think we need more than just auto. I'm not sure how much more, but I think having just auto would be too limiting*".

John Spicer's followup to those thoughts (in a private email, included with his permission) supports our claim that C++ users would benefit from the full template parameter list syntax. He also recommends a strategy (omitting the `typename` or `class` specifier for type *template-parameters*) for making the syntax more concise (which we favor, and should perhaps be a separate paper since it could apply to function templates):

> *I'd like to add that on further thought, generic lambdas should probably support the full template parameter list syntax.*
>
> *Most lambdas can probably get away with auto, but for the cases that can't saying "typename T", is not that more than just "T", and I think it would be a mistake to prohibit nontype template parameter and template template parameters from generic lambdas.*
>
> *Of course, we could allow both "<T1, T2>" and "<typename T1, typename T2>". If we do so, we should probably make that syntax available for normal templates too.*
>
> *It is currently the case that "typename identifier" is at type parameter and "typename nested-name-specifier identifier" is an unnamed nontype parameter. The same rule could be applied to cases without the "typename" (i.e., <T1> always declares a type template parameter named T1, never a nontype parameter of type T1 (if a type T1 is visible to normal lookup)).*

The *full template parameter list* syntax feature has been **implemented**,

[1] SF = Strongly Favor, F = Weakly Favor, N = Neutral, A = Weakly Against, SA = Strongly Against

2.2 Permit a lambda body to be an expression

Experience has shown that many lambda bodies are short, and of the form "`{ return` *`expression`*`; }`". Since we share Bjarne Stroustrup's view that there is value to terseness with lambdas[2], as a further convenience, we propose allowing such a lambda body to be written as simply "*expression*" with identical semantics:

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

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

Note: this extension would also apply to non-generic lambdas.

Since the *expression-body* form behaves similar to the *compound-statement-body* form with a single return statement containing the *expression*, we do not expect the return type deduction for such lambdas, or any generic lambdas to occur within a SFINAE (Substitution Failure Is Not An Error) context. This would be consistent with Merrill's (N3386) return type deduction for functions proposal. If SFINAE is desired here, it should be a separate proposal.

In addition, as a consequence of the grammar of an *expression* (note: we are actually proposing that the syntactic form actually be an *assignment-expression* that precludes *commas*, please read on for the rationale), ambiguities can arise in certain contexts:

1. In the setting of a *mutable* qualifier:

```
int x = 10;
[=](auto a) mutable a + ++x;
```

*We note no ambiguities here***.** While the construct may look initially odd, we suspect that once a reader is used to seeing this in code, it would not introduce much complexity.

2. In the setting of a *trailing-return-type* or an *exception-specification*:

```
[](auto a) -> int* a;              // 1 -- ok
[](auto a) -> int* (a);            // 2 -- What to do here?
[]<bool B>(auto) noexcept (B);     // 3 -- What to do here?
```

When parsing the trailing return type or the exception-specification, in the setting of an expression-body the parentheses could signify a function type, a constant-expression being passed to noexcept, or the beginning of an expression. One could institute new rules to try and resolve these ambiguities

[2] *"...If you see a lambda as something that typically appears in the middle of an expression, the amount of typing needed to define it becomes important. Thus Andrew and I tend to focus on the ultra-terse [terse concepts-lite example]..."*

(the solution space here might overlap with the one employed with the *new* expression i.e. *new-type-id* and *new-initializer*), *but it would be simpler to forbid trailing return types and exception-specifications if the expression-body form of the lambda is being used.* If this restriction is noted to be too draconian in the future, efforts can then be taken to try and resolve this ambiguity. We are recommending a conservative approach, and this may prove too conservative for some developers, therefore before we invest further resources in a solution, at this time we are simply requesting guidance from the EWG on this matter.

3. When a lambda expression is used within the context of declaring a function (default arguments) or calling a function or any other context in which it is followed by a *comma*:

```
template<class T, class U> void foo(T t, U u);      // 1

foo([](auto a) a, [](auto b) b);                    // 2


template<class R, class A>
  void foo_def(R (*fp1)(A),                    // 3
       R (*fp2)(A) = [](auto a) a,
       R (*fp3)(A) = [](auto b) b);

 foo_def((int (*)(int))[](auto b) b);          // 4
```

We feel that the ambiguity of whether the *comma* that follows the *expression* is part of the *expression* (i.e. a comma separated list of expressions) or the function declarator should be resolved such that *the comma is NOT part of the expression. If a comma separated sequence of expressions is required, they can be enclosed in parentheses.* This can be specified using the syntactic form *assignment-expression* in the expression-body form (instead of the syntactic form *expression* which allows commas in the absence of being enclosed by parentheses*)* which would give us the desired results.

4. In the setting of an attribute-specifier-seq

We note no ambiguities here.

5. In defining a lambda without a lambda declarator

```
int local = 10;
[&] ++local ; // ok - but we require [&]() ++local;
```

While we note no ambiguities here currently, we do recommend that a lambda-declarator be required with the expression form, otherwise, should names ever be allowed in lambda expressions, we may end up having to confront the following ambiguity:

```
auto F1 = []() { return [](auto a) { return a; }; };

auto L = [] F1()(F1); // If named lambdas were ever allowed, is this
                      // a Lambda named F1 (with (F1) as body),
                      // or unnamed lambda with F1()(F1) as a body?
```

Therefore, the grammar we would propose (to prohibit the comma from being parsed as the body of the lambda expression without explicit parentheses) would be along the lines:

lambda-expression:
   lambda-introducer lambda-declarator$_{opt}$ *compound-statement*
   lambda-introducer lambda-declarator$_{opt}$ *assignment-expression*

As a consequence the following constructs would behave as so:

```
template<class T, class U> void foo(T t, U u);

// well formed, two lambdas passed to foo
foo([](auto a) a, [](auto b) b);

// no ambiguity with lambdas as default arguments
template<class R, class A>
  void foo_def(R (*fp1)(A),
       R (*fp2)(A) = [](auto a) a,
       R (*fp3)(A) = [](auto b) b);

 // ill-formed: 'b' is a separate variable with no initializer
auto M = [](auto a, auto b) a, b;

// ill-formed: two separate variables M and 'b' of different types
auto M = [](auto a, auto b) a, b = 10;

// well-formed: (a, b) is the body of the lambda
auto M = [](auto a, auto b) (a, b);


// well formed
int local = 10;
auto Increment = [&] ++local; // [&] { return ++local; }
Increment(); // local is now 11


// ill-formed
[]() ;       // no body
[](++local); // paren, expects parameter
```

All the ambiguities mentioned on the EWG reflector (messages 14111-14139) would be answered by resorting to the above grammar and restrictions mentioned above.

It is worth noting that intial user experience is showing that the expression-body form has strong appeal and can add to clarity of code. In addition, one of the authors who was initially neutral on this feature, after having written code to test the implementation, is now strongly in favor of this feature. Since nested lambdas are not uncommon, and with the advent of C++11, currying (transforming a function that takes multiple arguments in such a way that it can be called as a chain of functions, each with a single argument) with non-generic lambdas is becoming popular in C++ code, it is worth considering the readability issue and the lispy brace-like problem that can manifest in the absence of the expression-body form. Consider:

```
// curry3 is a lambda that when called with a functor (aka 'f')
// returns a lambda that if passed the first argument (aka 'a') to 'f',
// returns a lambda that if passed the second argument (aka 'b') to 'f'
// returns a lambda that if passed the third argument (aka 'c') to 'f'
// returns the result of calling 'f' with 'a', 'b', 'c'.

auto curry3 = [](auto f)
                [=](auto a)
                  [=] (auto b)
                     [=] (auto c) f(a, b, c);

auto sum = [](auto a, auto b, auto c) a + b + c;

auto val = curry3(sum)(1)(2)(3); // val = 1 + 2 + 3
```

Compare the above to the more verbose synonymous code:

```
auto curry3 = [](auto f) {
                return [=](auto a)  {
                  return [=] (auto b) {
                    return [=] (auto c) {
                         return f(a, b, c);
                    }}}};

auto sum = [](auto a, auto b, auto c)
           { return a + b + c; };

auto val = curry3(sum)(1)(2)(3);
```

While readability is no doubt subjective, it is our opinion that the expression-body form – if used judiciously – can enhance the clarity of code.

This feature has been **implemented**,

2.3 Allow auto forms in the trailing return type

Occasionally we need the return type of a lambda to be deduced as a reference. There is no way to do this for a lambda (even under N3386). Therefore, we propose supporting *auto* forms (including *decltype(auto)*) in the trailing return type:

```
auto L = [=](auto f, auto n) -> auto& { return f(n); };
auto M = [=](auto f, auto n) -> auto* { return f(n); };
auto N = [=](auto f, auto n) -> auto  { return f(n); };
```

2.4 Allow generic lambdas with variadic auto parameters

Both Scott Prager [std-proposals] and Nikolay Ivchenkov [c++std-ext-14216] independently have requested that generic **auto** parameters support variadic syntax. Consider the following:

```
auto PrinterCurrier = [](auto printer) {
   return [=](auto&& ... a) {
      printer(a ...);
   };
};
```

2. 5 Lambda Syntax for Functions vs Named Lambdas vs use of auto in function parameters

While there is clear interest on this topic, unfortunately we have not had enough time to work further on this important feature. Based on some preliminary discussion on the reflector [c++std-ext-14220 - 14232] it seems that a separate paper thoroughly discussing the various design choices would be in the C++ community's best interest..

3 Further work

At this time we are simply seeking guidance from the EWG as to which of these features should we continue working on.

4 Acknowledgments

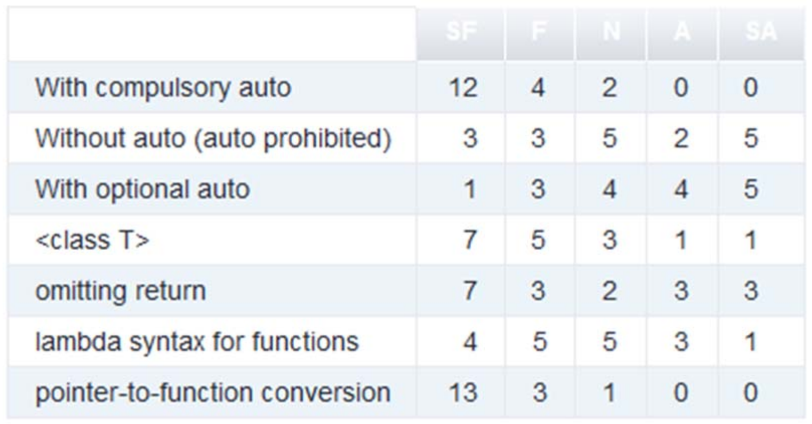
We are grateful for all the help and comments provided by those who participated in discussions on the various forums and provided us with valuable feedback, such as Richard Smith, John Spicer, Scott Prager (just to name a few).

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

Additional References:

- [Spicer-ext-14260] J Spicer. Post on the EWG Reflector
- N3386 J Merrill. *Return type deduction for normal functions*
- N3418 F Vali, H Sutter, D Abrahams. *Proposal for Generic Lambda Expressions*
- N3559 F Vali, H Sutter, D Abrahams. *Proposal for Generic Lambda Expressions (Rev 2)*

Portland 2012 EWG Voting Results:



	SF	F	N	A	SA
With compulsory auto	12	4	2	0	0
Without auto (auto prohibited)	3	3	5	2	5
With optional auto	1	3	4	4	5
<class T>	7	5	3	1	1
omitting return	7	3	2	3	3
lambda syntax for functions	4	5	5	3	1
pointer-to-function conversion	13	3	1	0	0