Proposal for Generic (Polymorphic) Lambda Expressions

(Revision 2)

Document no: N3559

Faisal Vali Herb Sutter Dave Abrahams

2013-03-17

Abstract

This document revises N3418: *Proposal for Generic (Polymorphic) Lambda Expressions* and incorporates feedback from the Evolution Working Group (Portland, October 2012). In this revision we propose the two features that received no opposing votes from the EWG (with the other features described in a separate document). Specifically, we propose that **`auto`** be required in a *lambda-expression's parameter-declaration-clause* to identify a generic lambda; and that a generic lambda with no *lambda-captures* contain a conversion function template to an appropriate *pointer-to-function*. After a brief discussion of the features (with details deferred to the Appendices), we describe standard wording. All the features proposed in this document have been implemented using clang.

0 Changes

The main differences between the initial proposal (N3418) and this document are:

– We only included those features that received no *against* votes

– **`auto`** is now required as a placeholder per Portland EWG direction

– Jason Merill's Return Type Deduction Proposal (N3386) that was accepted by the EWG, is referred to where needed to describe semantics

– We refer to our implementation experience using a fork (2012-07-11) of clang

– We refer to new user experience

1 Introduction

This document proposes generic lambda expressions and revises N3418: Proposal for Generic (Polymorphic) Lambda Expressions, which should be referenced for background and motivation. Today's C++11 lambda expression concisely creates an instance of a class having a non-template function call operator. We propose a pure extension of C++11 lambda syntax that creates an instance of a class having a function call operator template.

Here follow some examples of the proposed syntax extension in use:

```
// 'Identity' is a lambda that accepts an argument of any type and
// returns the value of its parameter.
auto Identity = [](auto a) { return a; };
int three = Identity(3);
char const* hello = Identity("hello");

// Conversion to function pointer for capture-less lambdas
int (*fpi)(int) = Identity;
char (*fpc)(char) = Identity;
```

2 Proposal

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

1 Allow **auto** *type-specifier* to indicate a generic lambda parameter
2 Allow conversion from a capture-less generic lambda to an appropriate pointer-to-function

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

2.1 Allow auto *type-specifier* to indicate a generic lambda parameter

If the initial *type-specifier* within the *decl-specifier-seq* of a lambda's *parameter-declaration* is **auto**, the expression creates a generic lambda 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 parameters, each use of **auto** is replaced by a unique template type parameter, which is added to the operator's *template-parameter-list*.

For example, this generic *lambda-expression* containing statement:

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

might result in the creation of a closure type, and object that behaves similar to the *struct* below:

```
struct /* anonymous */
{
    template <typename T, typename U>
    auto operator()(const T& x, U& y) const // N3386 Return type deduction
    { return x + y; }
} L;
```

In our initial proposal, we had recommended omitting **auto**, similar to syntax supported by other statically typed programming languages (D, C#, Java). A straw poll conducted by the EWG on 2012-10-16 showed that of the eighteen people voting, the majority either strongly favored (12/18) or favored (4/18) the mandatory use of **auto,** with no votes against or strongly against. On the other hand, omitting **auto** or making it optional had five out of eighteen members voting strongly against. Therefore we now propose that **auto** be mandatory. We recognize that **auto** can be replaced in the future by a placeholder (for e.g. '+') if further brevity is deemed necessary, but we do not propose this at this time. For further discussion regarding this decision, see Appendix B.

2.2 Allow conversion from a capture-less generic lambda to an appropriate pointer-to-function

A generic lambda with no *lambda-captures* shall have a public conversion function template to a pointer-to-function. The conversion function template shall be non-virtual, non-explicit and const (similar to the one for non-generic lambdas). It shall return the address of a function that, when invoked, has the same effect as invoking the generic lambda's function call operator with arguments of the same type as the type of the parameters of the function-pointer being initialized by the generic lambda.

For example, this generic *lambda-expression* containing initialization:

```
int (*fp)(int, char) = [](auto a, auto b){ return a + b; };
```

might result in the creation of a closure type, and object that behaves similar to the *struct* below:

```
struct /* anonymous */
{
   template<class A, class B>
      auto operator()(A a, B b) const  // N3386 Return Type Deduction
    { return a + b; }

  private:
    // Note: We don't want to simply forward the call to operator()
    //       since forwarding is not entirely transparent, and could
    //       introduce visible side-effects. To produce the
    //       desired semantics we copy the parameter-clause
    //       and body exactly
    template<class A, class B>
      static auto __invoke(A a, B b) {  // N3386 Return Type Deduction
        return a + b;
    }

    template<class A, class B, class R>
       using fptr_t = R (*) (A, B);

  public:
    template<class A, class B, class R>
      operator fptr_t<R, A, B>() const {
         return &__invoke;
    }
} L;

int (*fp)(int, char) = L;
```

Template Argument Deduction is used to deduce the *template-arguments* for the conversion function template when a pointer-to-function is initialized with a generic lambda. In returning an address to a function, the instantiation of the corresponding body of the function call operator template specialization occurs.

3 Implementation Experience

Generic Lambda expressions as described in this document have been implemented using a fork (2012-11-07) of clang with commits posted on github.

4 User Experience

Our Test Suite and User Experience of the features proposed in this document (and other generic lambda extensions) can be found at the following links:

- https://github.com/faisalv/clang-glambda/tree/master/test/CXX/generic-lambdas
- https://gist.github.com/4347130
- http://yapb-soc.blogspot.com/2012/12/clang-and-generic-polymorphic-lambdas.html

5 Changes to the Working Draft

Change in 7.1.6.4 dcl.spec.auto paragraph 1:

The **auto** *type-specifier* signifies that the type of a variable being declared shall be deduced from its initializer or that a function declarator shall include a *trailing-return-type* or that a lambda is a generic lambda.

Change in 7.1.6.4 dcl.spec.auto, Add after paragraph 2:

Otherwise, if the **auto** *type-specifier* appears as one of the *decl-specifiers* in the *decl-specifier-seq* of a *parameter-declaration* of a *lambda-expression*, the lambda is a *generic lambda* (5.1.2 expr.prim.lambda).

Change in 5.1.2 expr.prim.lambda paragraph 5:

The closure type for a non-generic *lambda-expression* has a public inline function call operator (13.5.4) whose parameters and return type are described by the *lambda-expression's parameter-declaration-clause* and *trailing return-type* respectively. For a generic lambda, the closure type has a public inline function call operator member template (14.5.2 temp.mem) whose *template-parameter-list* consists of one invented type *template-parameter* for each occurrence of **auto** in the lambda's *parameter-declaration-clause*, in order of appearance. The return type and function parameters of the function call operator template are derived from the *lambda-expression's trailing return-type* and *parameter-declaration-clause* by replacing each occurrence of **auto** in the *decl-specifiers* by the name of the corresponding invented *template-parameter*. This function call operator or operator template is declared **const** (9.3.1) if and only if the *lambda expression*'s *parameter-declaration-clause* is not followed by **mutable**. It is neither **virtual** nor declared **volatile**. Default arguments (8.3.6) shall not be specified in the *parameter-declaration-clause* of a *lambda declarator*. Any *exception-specification* specified on a *lambda-expression* applies to the corresponding function call operator or operator template. An *attribute-specifier-seq* in a *lambda-declarator* appertains to the type of the corresponding function call operator or operator template. [Note: Names referenced in the *lambda-declarator* are looked up in the context in which the lambda-expression appears. —end note]

Change in 5.1.2 expr.prim.lambda paragraph 6:

The closure type for a non-generic *lambda-expression* with no *lambda-capture* has a **public** non-virtual non-explicit **const** conversion function to ~~pointer to function~~ pointer-to-function having the same parameter and return types as the closure type's function call operator. The value returned by this conversion function shall be the address of a function that, when invoked, has the same effect as invoking the closure type's function call operator. For a generic lambda with no *lambda-capture*, the closure type has a **public** non-virtual non-explicit **const** conversion function template to pointer-to-function. The conversion function template has the same invented *template-parameter-list*, and the pointer-to-function has the same parameter and return types as the function call operator template. If the generic lambda has no *trailing-return-type*, deduction of the

pointer-to-function (14.8.2.3 temp.deduct.conv) shall behave as if a type *template-parameter* used to specify the return type of the pointer-to-function was added to the end of the conversion function template's *template-parameter-list* and used during template argument deduction. The value returned by any given specialization of this conversion function template shall be the address of a function that, when invoked, has the same effect as invoking the generic lambda's corresponding function call operator template specialization. [*Note*: This will result in the implicit instantiation of the generic lambda's body. Return type deduction (5.1.2 expr.prim.lambda) is done if no *trailing-return-type* is specified. The instantiated generic lambda's return type and parameter types shall match the deduced return type and deduced parameter types of the pointer-to-function — *end note*] [*Example:*

```
  // ill-formed: int (char) does not match char (char)
  int (*fp)(char) = [](auto a) { return a; };
```

— *end example]*

Change in 5.1.2 expr.prim.lambda paragraph 11:

If a *lambda-expression* has an associated *capture-default* and its *compound-statement* odr-uses (3.2) **`this`** or a variable with automatic storage duration and the odr-used entity is not explicitly captured, then the odr-used entity is said to be implicitly captured; Additionally, for a generic lambda, if its *compound-statement* names a variable with automatic storage duration in a *potentially-evaluated* expression (3.2 basic.def.odr) where the enclosing full-expression is dependent (14.6.2 temp.dep) on a generic lambda parameter and that variable is not explicitly captured, then the variable is said to be implicitly captured; all such implicitly captured entities shall be declared within the reaching scope of the lambda expression. [Note: The implicit capture of an entity by a nested *lambda-expression* can cause its implicit capture by the containing *lambda-expression* (see below). Implicit odr-uses of **`this`** can result in implicit capture. — end note]

Change in 5.1.2 expr.prim.lambda paragraph 12:

An entity is captured if it is captured explicitly or implicitly. An entity captured by a *lambda-expression* is odr-used (3.2) in the scope containing the *lambda-expression*. If **`this`** is captured by a local lambda expression, its nearest enclosing function shall be a non-static member function. If a *lambda-expression* odr-uses (3.2) **`this`** or a variable with automatic storage duration from its reaching scope, that entity shall be captured by the *lambda-expression*. Additionally, for a generic lambda, if its *compound-statement* names a variable with automatic storage duration in a *potentially-evaluated* expression (3.2 basic.def.odr) where the enclosing full-expression is dependent (14.6.2 temp.dep) on a generic lambda parameter, that variable shall be captured by the *lambda-expression.* If a *lambda-expression* captures an entity and that entity is not defined or captured in the immediately enclosing lambda expression or function, the program is ill-formed.

Change in 14.5.2 temp.mem paragraph 2:

A local class of non-closure type shall not have member templates.

6 Acknowledgments

We thank all those who read initial versions of this draft, who participated in discussions on the various forums and commented on it.

In addition we would like to thank:

> Jens Maurer for his indispensable assistance. Without him, this proposal would have been less polished and more solecistic; it would also have lacked standard wording.
>
> Doug Gregor for walking us through the Dimholt road and past the shadowy paradoxes that once haunted *lambda captures*.
>
> Scott Prager was instrumental in testing, discovering bugs and providing active feedback on our implementation. His blog article on our implementation of generic lambdas is insightful and thought provoking.

Adam Butcher is the author of the GCC patch mentioned in the earlier revision of this paper [He was incorrectly identified as Arthur Butcher].

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.

7 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
- N3418: F Vali, H Sutter, D Abrahams. *Proposal for Generic (Polymorphic) Lambda Expressions*
- N3386: J Merrill. Return type deduction for normal functions

Appendix A: The *Gregor Capture Paradox – template Variation*[1] and Name Resolution Issues

While in Portland, right after the Generic Lambda Discussion, Doug Gregor devised an interesting paradox that could arise with generic lambdas, which upon our request, he emailed to us (included with his permission):

Given this code:

```
template<bool C, typename T, typename U> struct if_;

template<typename T, typename U> struct if_<false, T, U> {
  typedef U type;
};

template<typename T, typename U> struct if_<true, T, U> {
  typedef T type;
};

struct X {
  X(int);
  operator int() const;
};

template<bool C>
constexpr typename if_<C, int, X>::type
f(typename if_<C, int, const int&>::type arg) { return arg; }

void test() {
  const int x = 17;

  auto g = [](auto a) {
       const int i = f<sizeof(a) == sizeof(char)>(x);
  };
  //  g('a'); // okay: does not capture x
  //  g(17);  // error: captures x
}
```

Whether the auto assigned to 'g' is well-formed or not depends on whether 'x' needs to be captured. Unfortunately, whether x needs to be captured depends on the type of 'a', which can vary from one use to another. That's not implementable, given that the capture list needs to be determined when the lambda is created and initialized, long before it is invoked.

[1] Not to be confused with the Gregor Capture Paradox – typeid Variation [c++std-core-21303], Core Issue 1468

> *The suggested resolution would be for generic lambdas (or all lambdas?) to capture any local variable from an enclosing scope that is named within the body of the lambda, rather than only capturing those local variables from an enclosing scope that are odr-used within the body. That would make this code always ill-formed, but if one turned the lambda into [=] or [&], it would work fine (by always capturing 'x').*

After a private email exchange between Doug and us, we all agreed that an initial reasonable resolution might be:

> ***generic lambdas to capture any local variable from an enclosing scope that is named in a potentially-evaluated context within the body of the lambda.***

In our implementation, Doug's example results in the following error:

```
test-doug.cpp:41:48: error: variable 'x' cannot be implicitly
captured in a lambda with no capture-default specified
       const int i = f<sizeof(a) == sizeof(char)>(x);
                                                  ^
test-doug.cpp:37:12: note: 'x' declared here
 const int x = 17;
           ^
test-doug.cpp:39:11: note: lambda expression begins here
 auto g = [](auto a) {
```

It compiles fine if a default capture (or an explicit capture of 'x') is specified.

Since only captureless lambdas have a conversion to a pointer to a function, it might be useful to permit as many possible captureless lambdas that we can reliably and unambiguously identify at definition time:
For e.g consider:

```
void f2(int i) { }
void test() {
  const int x = 17;

  auto g = [](auto a) {
       const int i = f2(x);
  };
  g('a'); // okay: does not capture x in our implementation
}
```

Therefore it might be worthwhile to consider the following notion:

> ***generic lambdas to capture any local variable from an enclosing scope that is named in a potentially-evaluated context, and can not unambiguously be resolved as non odr-used at the point of definition of the lambda within the body of the lambda.***

It is also worth reminding readers that dependent name resolution in generic lambdas would work as expected. Consider:

```
template<class T> void f_ADL(T t) {
  cout << "f_ADL(T)\n";
}
auto test2 = []() {
 auto g = [](auto a) {
       f_ADL(a);
 };
 return g;
};

struct Y { };

void f_ADL(Y y) {
  cout << "f_ADL(Y)\n";
}

int main() {
    auto g = test2();
    g('a');
    g(Y()); // ADL at instantiation-context finds f_ADL(Y)
}
```

As one would expect with dependent name resolution within templates, our implementation prints:

```
f_ADL(T)
f_ADL(Y)
```

As required by the standard, if two different points of instantiation result in a violation of the ODR rule, the program is ill-formed, no diagnostic required.

Appendix B: Discussion regarding omitting `auto`

It seems the main arguments against omitting auto are:
1 Ambiguity concerns (since identifiers can be omitted in lambda parameters)
2 Readability concerns
3 Stylistic concerns (in the setting of Concepts [still being actively designed])

While reasonable people will disagree about the readability of code that omits **auto**, we feel the issue of ambiguity (does the parameter represent the name of a generic lambda parameter, or the type of a non-generic parameter with its name omitted?) deserves attention. Consider the following example posted on 2012-12-20 by Nevin Liber on std-proposals (which we have modified only to give the lambda a name by inserting the line '*auto schrodinger =*' below):

```
struct Missiles
{
       Missiles(int numberToCreate) { /*... */ }
       Missiles& operator()(int numberToLaunch) { /* ... */ }
};

auto schrodinger =
  [](Missiles){ return Missiles(1); };

This behavior has changed silently under the proposal.
```

In C++11, the variable *schrodinger* above unambiguously represents a non-generic lambda with an unnamed parameter of type *Missiles* that returns a value of type *Missiles*. If **auto** is allowed to be omitted, an interesting superposition might arise: Is *schrodinger* a generic lambda, a non-generic lambda, or an error (or all three until actually invoked *quantum-physics-humor-alert* ;)?

If the definition of *struct Missiles* is omitted, *schrodinger* would represent a generic lambda – this much is clear. What is less clear is what *schrodinger* should represent if *struct Missiles* is not omitted (and **auto** is not required to indicate a generic lambda parameter). While one could try and formulate rules to disambiguate the situation (for e.g. a generic parameter shall not be named the same name as a type-name that can be found in its context), and while there is precedent in C++ for entangled code (consider: `int *a; struct a { }; int (*fp)(a); )` we feel that introducing further potential for such nuanced entanglement might increase complexity while adding little benefit (considering there was significant disagreement about the wins of omitting **auto**). Therefore, as the EWG favored in Portland, we now propose that **auto** be required.

Interestingly, a suggestion has been made by Richard Smith on std-proposals that we consider reversing the decision that led to the variable identifier being optional in a non-generic lambda parameter. We note that this would avoid the above ambiguity, and we would be in favor of this, if the rest of the EWG found the solution palatable. This

direction can also be pursued at a later date, but it might be best to resolve this sooner than later.

Appendix C: Inconsistency between auto variable deduction and generic parameter deduction

It is worth noting that since **auto** lambda parameters are converted into invented *template-parameters* and added to the *template-parameter-list* of the generic lambda's function call operator template, the semantics of **auto** lambda parameters is defined entirely in terms of deducing template arguments from a function call (temp.deduct.call, 14.8.2.1) and not **auto** deduction semantics (dcl.spec.auto, 7.1.6.4 para 6). This is only relevant when deducing from a *braced-init-list*, since in all other regards, the deduction semantics are identical. In the setting of a *braced-init-list,* **auto** deduction replaces: `auto IL = { 1, 2, 3} with std::initializer_list<T> IL = { 1, 2, 3 }` and then deduces against T. In the setting of deducing template arguments from a function call, this does not work.

For e.g (in our implementation)

```
auto IL = { 1, 2, 3};                    // ok
([](auto IL) { return 3; })({1, 2, 3});  // error
```

We recognize that this is confusing, and propose that a paper be written to unify these semantics.