Lambda expressions and closures for C++

Document no: N1968=06-0038

Jeremiah Willcock Jaakko Järvi Doug Gregor Bjarne Stroustrup
Andrew Lumsdaine

2006-02-26

Abstract

This proposal describes a design for direct support for lambda expressions in C++. The design space for lambda expressions is large, and involves many tradeoffs. We include a thorough discussion of the benefits and the drawbacks of our design. In addition, we describe several other viable alternatives that warrant consideration.

1 Introduction

Many programming languages offer support for defining local unnamed functions on-the-fly inside a function or an expression. These languages include Java, with its *inner classes* [GJSB05]; C# 3.0 [Csh05]; Python [Fou05]; ECMAScript [ECM99]; and practically all functional programming languages, Haskell [PH+99] and Scheme [ADH+98]. Such functions, often referred to as *lambda functions*, or *closures*, have many uses: as the arguments to higher-order functions (such as std::for_each in the context of C++), as callbacks for I/O functions or GUI objects, and so forth. This document discusses the design space of closures for C++, and suggests a possible specification for their syntax and semantics, and outlines a possible implementation for the specification.

We use the following terminology in this document:

- **Lambda expression** An expression that specifies an anonymous function object.
- **Lambda function** This term is used interchangeably with the term "lambda expression."
- **Closure** An anonymous function object that is created automatically by the compiler as the result of a *lambda expression*. A closure stores those variables from the scope of the definition of the lambda expression that are used in the lambda expression.

A lambda expression defines an object — not just a function without a name. In addition to its own function parameters, a lambda expression can refer to local variables in the scope of its definition. Thus, a lambda expression defines a value that consists of both code as well as the values of the variables referred to in the code — a closure. Such closures have a natural representation as C++ function objects, as discussed in Section 3.

Even though hand-written function objects can be used in place of the proposed closures, the direct definition and construction of function objects is cumbersome. To define a function object, an entire class must be defined, in many cases including member variables and a constructor to initialize them. Furthermore, the class must be named prior to its use, and defined outside of the function that uses it (unless proposal N1427 [Wil03], to allow local classes as template arguments, is approved). The following code defines a function object type (and corresponding object) that stores a copy of the local variable i, and passes the object to an Standard Library algorithm:

```
struct less_than_i {
  int i;
  less_than_i(int i) : i(i) {}
  bool operator()(int x) const {return x < i;}
};

int i = 17;
find_if(v.begin(), v.end(), less_than_i(i));
```

The lambda expressions described in this proposal are syntactic sugar for defining function object classes, such as the above, and constructing function objects from them using the contents of (or references to) local variables.

1.1 Motivation

The lack of a syntactically light-weight way to define simple function objects is a hindrance to the effective use of several generic algorithms in the Standard Library. As an example, consider the above call to the find_if() function. The definition of the function object is so verbose that it would be much easier (and likely clearer to readers of the code) to write a **for** loop to iterate through a sequence than to define a new function object type, construct an object using the correct arguments, and then call find_if() using it. Similar arguments apply to other simple Standard Library functions, such as for_each() and transform(). This is not obvious from textbook examples of Standard Library use — in simple cases, such as the example above, the standard function objects and binders suffice:

```
int i = 17;
find_if(v.begin(), v.end(), bind2nd(less<int>(), i));
```

Though a bit clumsy, the syntax is relatively concise. However, many current C++ programmers would need to bring out their manuals to find the meanings of bind2nd and less<**int**>.

The bind library in TR1 introduces a more expressive partial function application mechanism than std::bind1st and std::bind2nd, expanding the set of cases where explicit definition of function objects can be avoided. Furthermore, libraries such as the Boost Lambda Library [JP02], FC++ [MS04], and Phoenix [de 02] more or less define little sublanguages for writing lambda expressions. For example, using the Boost Lambda Library, the above example can be written as:

```
int i = 17;
find_if(v.begin(), v.end(), _1 < i);
```

This is actually very close to an ideal: it tersely states the operation to be done in a form similar to what would be written in the body of an explicit loop. Unfortunately, this elegant solution suffers from serious problems in terms of usability, implementation complexity, and performance. It also introduces a "magic" meaning to _1: the first argument to the function object (closure) generated from _1 < i. Note that the type of _1 is deduced from the context of the call: whatever is used by find_if() as an argument to **operator**()() on its predicate is compared to i using <. This differs from the way an explicitly written function object (whose **operator**()() is not a template) is defined; in that case the argument is coerced into the type expected by **operator**()(). For example, in the class less_than_i the stored value (corresponding to i in the example above) is always compared to an **int**.

1.1.1 Library problems

There are significant limitations in the library-based solutions. For example:

- A lambda expression using one of these libraries must be written as an expression using user-defined operators. This rules out using normal C++ syntax for function calls, variable declarations, or control structures. Thus, all but the simplest lambda expressions look unnatural when written using a library-based solution.
- Compilation times increase substantially when using a lambda library as opposed to a hand-written function object, and run-time performance can suffer if the compiler fails to fully inline the large number of intermediate function calls common to these libraries.
- Erroneous uses of lambda libraries typically manifest in extremely long error messages.
- Most importantly, there are many common situations where the behavior of lambda libraries is very difficult for users to understand, and common situations where the libraries do not work at all.

Regarding the last point, based on abundant feedback from the users of the Boost Lambda and Bind libraries (see Figure 1 for an example), it is clearly difficult for programmers to grasp what these situations are, and a lot of development time is wasted in trying to bend the libraries to do what they cannot do. For example, Figure 1 shows an attempt to capture a member function call in a lambda function, where the object argument is a parameter of the lambda function: _1.size(). This syntax follows naturally from how the rest of the library works, but cannot be supported. A library such as tr1::bind would allow this kind of lambda expression to be emulated as bind(&list<list<**int**> >::size, _1). As one can see, this is quite verbose.

The verbosity is worse with overloaded functions or function templates. For example, assume a member function foo() of some class X. If foo() is overloaded, &X::foo requires a cast (to a member function pointer type, not often used by beginning C++ programmers) to select the correct member function; if foo() is a member template, &X::foo is not a value at all, and thus foo() would need to be explicitly instantiated prior to taking its address. The code in these cases becomes hard to read, and it is not easy to understand the reasons for the explicit instantiations or casts, and thus when they are needed. The following two examples demonstrate this problem; the first assumes foo() is a two-argument overloaded member function, the second that it is a two-argument member function template with one type parameter:

I'm trying to use Boost.Lambda to count the total number of elements in a list of lists. This example shows what I'm trying to do:

```
int main()
{
    int count = 0;
    list<list<int> > ll;
    for_each(ll.begin(), ll.end(), var(count)+=(_1).size());
}
```

This will not compile:

(13): error C2039: 'size': is not a member of 'boost::lambda::lambda_functor<T>'

Is this possible to do?

Figure 1: A portion of a typical support query from a Boost Lambda Library user (shown here with the permission of the poster, Daniel Lidström). The erroneous part is the expression _1.size(), which cannot be supported by a lambda library.

```
bind((int(X::*)(int, double))(&X::foo), _1, _2, _3)
bind(&X::foo<int>, _1, _2, _3)
```

The confusion is amplified even further in the case of objects such as std::endl, which is a template but is not understood as one by most users. As the result, the first line below is a valid lambda expression, but the second line produces a compile-time error:

```
cout << _1 << "\n"
cout << _1 << endl // error
```

In addition to the problems related to expressing actions as lambda expressions, the library solutions are getting so complex that optimizers cannot manage to deliver acceptable performance. We have measured a simple algorithm (find_if() using a simple <) to take 2.5 times longer when using the Boost Lambda Library than using a hand-coded function object or a hand-coded loop. Lambda libraries also have scalability problems. Some use cases involve function objects which are too large and/or too complicated for a user-level library. There are a few options in those cases:

- Continue to use a lambda library, creating messy code for large lambda expressions where those libraries are inappropriate.
- Write function objects by hand.
- Write explicit loops, instead of using higher-order functions.

In sum, explicitly defined function object types can be used instead of the proposed lambda expressions, but they can be inconvenient to define. This leads people to experiment with increasingly elaborate techniques aimed at a more concise and conventional notation. These techniques often alleviate the verbosity, but are also themselves a source of complexity, confusion, and even other forms of verbosity.

What do we want? Something like:

```
find_if(v.begin(), v.end(), _1 < i);
```

is "the gold standard." We can compare this with the explicit **for** loop:

```
for (auto p = v.begin(); p != v.end(); ++p)
  if (*p < i) break;
```

This code communicates its intent less directly than the call to find_if(), but is significantly shorter than using find_if() with an explicit function object. Replacing a call to for_each() with a loop is even easier:

```
for (auto p = v.begin(); p != v.end(); ++p)
  cout << *p << endl;
```

With another planned C++0x feature, this becomes even more concise:

```
for_each (auto x : v)
  cout << x << endl;
```

Whatever lambda/closure mechanism we come up with, for it to be practical to use with the simpler Standard Library algorithms, it must be significantly less verbose than defining explicit function objects, not significantly more verbose than explicit loops, and clearer and more intuitive than explicit loops.

Importantly, the loop variants are trusted to be efficient. Library-generated function objects carry a run-time cost that is largely unknown. Their performance can be equivalent to that of loops, but that cannot be guaranteed for library implementations of lambda expressions. Thus, performance can be an argument for direct language support.

Obviously, loops are not a reasonable alternative to every use of lambda expressions. For callbacks in event-based systems or function objects used in complicated algorithms, such as sort(), the real alternative is hand-written function objects. For callbacks that are complicated (so that their specification is ugly if placed at the point of use) or reusable (so that their function object can be written once and used in several places), such explicit closures have an advantage.

We can summarize the design goals for C++ lambda functions as follows:

- **Efficiency** For many uses, lambda expressions and STL-style algorithms compete directly with explicit loops, where the body of the lambda expression is written as the body of the loop. Therefore, simple uses of a lambda expression with algorithms must equal their equivalent hand-written loops in time and space. In particular, inlining of simple lambda expressions is essential.
- **Notation** For many uses, lambda expressions and STL-style algorithms compete directly with explicit loops and with explicitly named function objects. Therefore, simple uses of a lambda expression with algorithms must not be significantly more verbose than their equivalent hand-written loops and hand-written function objects. Simple uses must be very simple.
- **Code sharing** Complicated operations are best designed for repeated use; that is, to be called from many places in a program. Thus, there is a distinct value to named, non-local functions and classes. Lambda expression are not intended to compete with such uses and there is no reason to make long lambda expressions particularly easy to write.
- **Implementation** The ideal is for a simple, transparent, translation model. Such a model would allow for simple implementations (both in the front end and the back end) and for users to easily comprehend both the meaning and the cost implications of uses of a lambda expression.
- **Generality** A lambda expression has aspects of a function (it performs an action) and an object (it has state). The primary aim is for lambda expressions to serve as "actions" for STL algorithms (the way function objects have traditionally been used) and similar "callback" mechanisms. The simplicity and efficiency of lambda expressions should not be compromised by attempting to generalize them beyond these uses.
- **Integration** The implementation of lambda functions must not complicate the definition of other C++0x features (such as concepts), must interact well with all C++ features (new and old), and not gratuitously duplicate existing features.

2 Proposal

We propose to extend the C++ language with *lambda expressions*, and define the semantics of these unnamed local functions via translation to *closures*: function objects implemented using local classes. The essence of our proposal is thus a concise syntax for defining a particular class of function objects. This translation directly suggests (but does not require) an implementation approach that retains the same performance as that of hand-written function objects.

The elements that define the behavior of a closure are:

- full signature of **operator**()(),
- the body of **operator**()(),
- types and names of member variables, and
- the expressions with which member variables are initialized.

Current C++ allows the programmer to control all these aspects, but with the verbose syntax of defining a class with its member variables, function call operator, and constructor, and then constructing an object of that type. We set our goal to be less verbose than that. The suggested design is thus the result of analyzing which of the above components can be given meaningful defaults and be left out when defining closures. We describe our current best effort in finding the sweet spot in this design space, and justify our decisions. We start from simple lambda expressions, where only

the function parameter list is relevant, and describe their translation, and end with lambda expressions where most of the above components need to be controlled.

The proposed lambda functions are *primary-expression*s, and thus can be written directly where they are used. The body of a lambda function can refer to local variables from its enclosing context, which the user can choose to store by reference or by copy in the newly created function object. Unlike in some library techniques, which support templated lambda functions, the proposed lambda functions are monomorphic: the parameter types must be defined and be non-generic. Thus, in the function object resulting from the translation, the function call operation is not a template. This is in some ways a step back from the library-based lambda expressions, which construct function objects with a templated function call operator. These polymorphic lambda functions, without significant additional machinery, would conflict with modular type-checking of concept-constrained generic functions. We justify the design choice of monomorphic lambda expressions in detail in Section 5.1.

Lambda functions are defined with the following syntax:

lambda-function:
  <> (*parameter-declaration-clause*) -> *type-id*
    local-var-clause$_{opt}$ *exception-specification*$_{opt}$ *compound-statement*
  <> (*parameter-declaration-clause*)
    local-var-clause$_{opt}$ *exception-specification*$_{opt}$ *compound-statement*

local-var-clause:
  extern (*local-var-list*$_{opt}$)

local-var-list:
  local-var
  local-var-list , *local-var*

local-var:
  identifier
  ∗**this**

A diamond (written as <>) introduces a lambda expression, and is followed by a standard function parameter list (possibly including default values for parameters). After this is optionally an arrow (written as −>) and the function return type, as in the **decltype/auto** proposal N1705 [JSD04]. If these are omitted the return type is deduced from the body of the function. Section 2.1 explains when this is possible; in the examples below we always specify the return type explicitly. After the return type follows an optional list of local variables (that need to be accessed by reference in the closure) in parentheses and preceded by the keyword **extern**. Note that reusing an existing keyword, which already has a different meaning in the language, may not be ideal. Other syntactic options to consider here might, for example, include punctuation characters. The special syntax ∗**this** is also allowed, indicating that the current object (if the enclosing context of the lambda expression is a member function) should be kept by pointer in the closure. The pointer **this** itself will always be copied into the closure, and by default the entire object will be copied, with the stored **this** pointer pointing to the new copy. The body of the lambda expression follows; as in any other function definition, the body is required to be enclosed in braces.

A lambda expression defines a new function object of unspecified type. This object has a compiler-generated copy constructor (if all stored local variables are copyable, just as for any other class type), a move constructor (if that proposal is approved, and if all stored variables are movable and/or copyable), and a function call operator whose parameter list and return type are those specified by the lambda expression.

As an example, consider the lambda expression:

```
⟨⟩(int x, int y) → int {return x + y;}
```

The behavior of this expression is the same as the behavior of the following class:

```
class new_class_name {
  public:
  int operator()(int x, int y) const {return x + y;}
};
... new_class_name() ...
```

If the proposed lambda functions are implemented via translation to function objects of a local class type, the new generated class is inserted immediately before the statement containing the lambda expression, and the construction of an object of that class replaces the lambda expression itself. For example, the code:

```
int i = 1, j = 2;
if (⟨⟩ (int x, int y) → int {return x + y;} (i, j) == 3) ...
```

thus has the same behavior as:

```
int i = 1, j = 2;
class new_class_name { /* definition as above */};
if (new_class_name()(i, j) == 3) ...
```

Note that the newly created closure is an rvalue, not an lvalue. Thus, it can be passed to functions which accept their arguments by copy, **const** reference, or rvalue reference (if that is approved). It cannot be passed to a function which takes its argument by non-**const** reference, even if this function is a template. Explicitly casting each closure to a **const** reference when it is constructed would allow this to work, but at the cost of forbidding move constructors from being used. At least one function in the Standard Library, random_shuffle(), accepts a function object by non-**const** reference; the rvalue reference proposal N1690 [HAD04] suggests changing this to an rvalue reference specifically to allow newly-constructed function objects to be used.

The body of the lambda expression can refer to the parameters of the lambda expression, global variables, and local variables from its enclosing expression context. In the case of local variables, copies of the values of these variables are automatically added as (hidden) member variables of the closure object. These member variables are **private** to the class, and their names are not visible except in the lambda expression body. For example, the following code uses a lambda expression which refers to a local variable (y) from the enclosing scope:

```
void f() {
  int y = 3;
  g(⟨⟩ (int x) → int {return x + y;});
}
```

Based on the translation semantics, this code has the same behavior as the following code:

```
void f() {
  int y = 3;
  class new_class_name {
    int y;
    public:
    new_class_name(const int& y): y(y) {}
    new_class_name(const new_class_name& o): y(o.y) {}
    int operator()(int x) const {return x + y;}
  };
  g(new_class_name(y));
}
```

Storing the variables as members in the closure allows them to be referenced from within the function call operator of the closure object, even if the object outlives the original local variables. Storing these members by copy differs from closures in languages, such as Scheme, where closures must effectively store references to local variables which are modified by the lambda expression. Scheme, however, is garbage collected. Without garbage collection, closures that store references to local variables can be dangerous, easily leading to dangling references, and thus undefined behavior. This happens if a closure lives longer than a local variable to which it stores a reference. For example, one can store a closure into a tr1::function defined outside of the scope where the closure is defined, and outside of the scope of the local variable. The following example demonstrates this behavior. Here we use special syntax **extern**(y) to force the closure to store a reference to y, instead of storing its value by copy:

```
tr1::function<int(int)> callback;
void f() {
  int y = 3;
  callback = ⟨⟩(int x) → int extern(y) {return x + y;};
}
...
f();
callback(5); // undefined behavior
```

Since the closure stores a reference to y, by the time f() returns, y's lifetime is over, but callback still holds a reference to it. Invoking (or even copying) callback thus leads to undefined behavior.

The following is another example of a callback which would contain a dangling reference, if references to local variables were stored instead of copies:

```
void init_gui() {
  ...
  label* l = new label("Hello");
  button* b = new button("Change label");
  b → set_on_push_callback(⟨⟩() → void {l → set_text("World");});
  ...
}
```

The pointer l is saved by copy inside the new closure, allowing the label's text to be changed by this callback even after the function init_gui() returns.

If a variable is stored in the closure by copy, each copy of the closure will have its own copy of the variable (with the same lifetime as that particular copy of the closure), and no dangling references to the variable can result. We thus make this safer design choice the default, and provide an explicit syntax for declaring that the closure should, instead of a copy, store a reference to a local variable. This is done by listing the variables following the **extern** keyword, as demonstrated by the above example.

In many cases, storing variables by reference in closures is desirable, or even necessary. In the following code, for example, each call to the lambda function modifies the local variable sum as a side-effect:

```
void f() {
  int sum = 0;
  for_each(a.begin(), a.end(),
    ⟨⟩(int x) → int extern(sum) {return sum += x;});
}
```

The semantically equivalent translated code for this example is:

```
void f() {
  int sum = 0;
  class new_class_name {
    int& sum; // reference, rather than const value
    public:
    new_class_name(int& sum): sum(sum) {}
    new_class_name(const new_class_name& o): sum(o.sum) {}
    int operator()(int x) const {return sum += x;}
  };
  for_each(a.begin(), a.end(), new_class_name(sum));
}
```

Note that local variables are stored as non-**mutable** members. This is to make code that tries to modify a local variable stored by copy to fail. Consider the above example again without the **extern** clause:

```
void f() {
  int sum = 0;
  for_each(a.begin(), a.end(),
    ⟨⟩(int x) → int {return sum += x;});
}
```

Now the sum member will not be **mutable** or a reference, and thus sum += x will generate a compile-time error (as the closure's **operator**()() is **const**). Storing sum as a **mutable** member would keep the code legal, but the assignments would modify the sum member variable in the closure, and their effect would not be observable outside of the closure. Thus, the code would compile but likely not behave as the programmer expected.

2.1 Omitting the return type

The return type of a lambda expression can be omitted if the body of the lambda function contains at most one return statement. If no return statements are present, the return type of a lambda expression is **void**, otherwise the return type is the **decltype** of the returned expression. For example, the lambda function:

```
⟨⟩(int x, double y) {return x + y;}
```

is semantically equivalent to the function:

⟨⟩(**int** x, **double** y) → **decltype**(x + y) {**return** x + y;}

2.2 Syntactic savings compared to function objects

We state in the introduction of this proposal that lambda expressions are essentially syntactic sugar for defining function object classes and constructing function objects. Comparing the proposed lambda expressions to explicitly writing function objects, we can observe the following syntactic "savings:"

- The closure object does not need to be explicitly constructed.
- The types of the members (referenced local variables) do not need to be specified.
- If members are stored as copies, their names do not need to be specified.
- The return type of the function call operator does not need to be specified in many cases.
- A constructor to initialize the members of the closure does not need to be defined.

3 Implementation issues

This proposal does not specify any particular implementation for lambda expressions or the closures they create. However, there is an intended implementation strategy which leads to lambda expressions with performance comparable to hand-written function objects, possibly at the expense of compilation time. In this strategy, each lambda expression becomes a new local class, along with the creation of a new object of this class. This translation assumes that references to references are legal (Core Defect 106 and proposal N1245 [Str00]), and that the lambda expression is not inside a member function:

```
void f() {
  func(
    ⟨⟩(type1 param1, type2 param2, ...) → return_type
    extern(local1, ...) { body which uses other local vars var1, var2, ...}):
}
// becomes
void f() {
  class __some_unique_name {
    const typename remove_reference<decltype(var1)>::type var1;
    const typename remove_reference<decltype(var2)>::type var2; ...
    decltype(local1)& local1; ...

    public: // But hidden from user
    __some_unique_name(const decltype(var1)& var1, const decltype(var2)& var2, ...,
                       decltype(local1)& local1, ...)
      : var1(var1), var2(var2), ..., local1(local1), ... {}

    public: // Accessible to user
    __some_unique_name(const __some_unique_name& o)
      : var1(o.var1), var2(o.var2), ..., local1(o.local1), ... {}

    return_type operator()(type1 param1, type2 param2, ...) const
      { body }
  };
  func(__some_unique_name(var1, var2, ..., local1, ...));
}
```

In this translation, __some_unique_name is a new name, not used elsewhere in the program in a way that would cause conflicts with its use as a closure type. This name, and the constructor for the class, do not need to be exposed to the user — the only features that the user can rely on in the closure type are a copy constructor (and a move constructor if that proposal is approved) and the function call operator. Closure types do not need default constructors, assignment operators, or any other means of access beyond function calls. It may be worthwhile for implementability to forbid creating derived classes from closure types. The metafunction remove_reference converts reference types to their underlying value types. This is done for consistency, so that members are stored in closures by reference only if explicitly requested using the **extern** variable list.

One potential issue is in a lambda expression inside a non-static member function: uses of the **this** keyword and unqualified accesses to member variables and functions must be changed to be relative to the enclosing class, not the newly created closure. The **this** pointer should effectively be treated as a local variable, and so must be stored in the closure object if it is used in the lambda expression's body. Also, the object pointed to by **this** will also be copied into the closure by default, and uses of **this** inside the lambda expression should point to that copy.

4 Relationship to other proposals

This proposal relies on other previously proposed language changes. First, allowing local classes to be used as template arguments, as proposed in N1427 [Wil03] enables a notably simpler implementation for lambda functions. The suggested implementation of a lambda expression is as a local class (see Section 3), and such classes must be allowed as template arguments for closures to be used as function objects in calls to generic algorithms. This should not be a major change to the language, other than that name mangling may need to be adjusted to create mangled names for local classes if they do not have them already. If local classes are not allowed as template arguments, lambda functions would need to be translated to classes defined in namespace scope, which is possible but more difficult, especially with nested lambda functions or when the enclosing function is a function template.

Lambda expressions as such are much more useful with the **decltype** extension proposed in N1705 [JSD04], and in particular, inferring the return type from the return expression is defined in terms of **decltype**. The function declaration syntax in which the return type follows the parameter list (separated by an arrow) also originates from that proposal. Deducing the type of a variable from its initializer expression, also in N1705, would allow lambda expressions, whose types are not otherwise named, to be stored directly into variables when they are defined.

The current proposal supersedes the bind function templates proposed in N1455 [DGJP03], and which are now part of TR1. The lambda expressions and closure objects provided in this proposal cover all features provided by N1455, with the exception of polymorphic lambda functions, and use much cleaner and more direct syntax to do so. A partial application of a function, as provided by tr1::bind, is a restricted form of a lambda expression, and full lambda expressions include all of the functionality of standard binders plus much more flexibility and safety. Moreover, complex expressions written using TR1 binders result in heavily nested template instantiations, which tend to consume a lot of compilation resources. On the other hand, binders do not require names or types to be given for their parameters (they are implicitly numbered in sequence and fully polymorphic), and have a syntax optimized to their specific application.

5 Design issues

There are several aspects of the design of this extension which are subject to debate. This section attempts to explain the various decisions which were made, as well as other alternatives which may be worth considering.

5.1 Why lambda functions are not templates?

We anticipate a very common use for lambda expressions to be as function objects to be passed to Standard Library algorithms. Libraries, such as the Boost Lambda Library [JP02], or tr1::bind in more limited cases, cater to this need. Though not a general solution, in the cases where these libraries apply, they provide two benefits over the solution proposed here:

1. **Very concise syntax**: Compare, for example, the following function written using the Boost Lambda Library:

   prod *= _1

   and with the syntax we propose (omitting the return type):

   ⟨⟩(**int** x) **extern**(prod) {prod *= x;}

2. **Polymorphism**: In the library-based lambda expressions, it is not necessary to specify the formal parameter types of the lambda function, or its return type. In particular, the parameter types in generic code may be quite verbose, and thus awkward. For example, making the multiply_list() function generic results in a fairly cumbersome lambda expression:

   template <**typename** InputIterator>
   typename iterator_traits<InputIterator>::value_type
   multiply_list(InputIterator begin, InputIterator end) {

```
  typename iterator_traits<InputIterator>::value_type prod = 1;
  std::for_each(begin, end,
                ⟨⟩(typename std::iterator_traits<InputIterator>::value_type x)
                  → void extern(prod) {prod *= x;});
}
```

Defining a **typedef** and omitting the return type can improve on this a little:

```
template <typename InputIterator>
typename iterator_traits<InputIterator>::value_type
multiply_list(InputIterator begin, InputIterator end) {
  typedef typename iterator_traits<InputIterator>::value_type vt;
  vt prod = 1;
  std::for_each(begin, end, ⟨⟩(vt x) extern(prod) {prod *= x;});
}
```

We considered the possibility of supporting polymorphic lambda functions, whose parameter types need not be specified. Such a lambda function would implicitly be an unconstrained template, accepting any argument types; the body would be checked against the particular argument types used when the call to the lambda function is instantiated. Such functions would not create significant difficulties for the proposed translation model. A lambda expression whose parameter types were not specified would simply be translated to a function object whose function call operator is a template. Such lambda functions would, however, clash with the modular type-checking which we attempt to introduce to C++ via concepts [SD05, GSW+05].

To show this, consider the following broken definition of a function multiply_list(), using a concept-enabled C++, and invented syntax for a polymorphic lambda function:

```
template <typename Iter> where {InputIterator<Iter>}
value_type multiply_list(Iter begin, Iter end) {
  value_type prod = 1;
  std::for_each(begin, end, ⟨⟩(x) extern(prod) {prod *= x;});
}
```

This definition is incorrect because there is no requirement that the value_type of the iterator supports an **operator***(). Without the lambda expression, for example using an explicit loop over the input range, this error would be caught before multiply_list() is instantiated. With a monomorphic lambda function:

```
⟨⟩(const value_type& x) extern(prod) {prod *= x;}
```

the error would also be caught before instantiation time, as the compiler would know that x has type value_type, which may not necessarily have an **operator***(). With the unconstrained polymorphic lambda function, however, the error in the definition of multiply_list() would not be caught until it is instantiated on a particular iterator type whose value type does not have **operator***() defined. This behavior occurs because the type of x is an unconstrained template parameter, and so all operations are allowed until the template is instantiated. Thus, implicitly polymorphic lambda expressions break the level of separate type checking provided by concepts, or at least, complicate separate type checking significantly. Explicitly polymorphic lambda expressions (with their own **where** clauses) would not have this problem, but would be very verbose, and the polymorphism they provide would only rarely be used.

Note that it would be possible to leave the parameter types of the lambda function unspecified, and try to deduce them from the use of the lambda function. In the above example, the lambda function is passed to the for_each() function, whose signature in concept-enabled C++ would constrain F, the type of the function object, to model Callable1<F, value_type>, where value_type is short for InputIterator<Iter>::value_type. Thus, the argument type value_type in constraint using the Callable1 concept would be used as the argument type of the lambda expression. This constraint would then be used to attempt to type-check the body of the lambda, and this check would fail. To make the code type-check, we could rewrite it as:

```
template <typename Iter> where {InputIterator<Iter>, Multipliable<value_type>}
value_type multiply_list(Iter begin, Iter end) {
  value_type prod = 1;
  std::for_each(begin, end, ⟨⟩(x) extern(prod) {prod *= x;});
}
```

Note, however, that the resulting closure passed to for_each() would not have a templated function call operator. This mechanism would thus only allow a more convenient way to define monomorphic lambda functions. We have not explored this approach in detail; it would at the least require special handling of the Callable* concepts in the compiler.

A possible argument in favor of providing polymorphic lambda functions is that the existing bind and lambda libraries only provide (unconstrained) polymorphic lambda functions. Although this is true, most actual uses of these libraries involve lambda functions which are only called on one particular set of argument types. The libraries allow polymorphism only because it would be very inconvenient for them to require parameter types to be given. A language-based lambda extension does not have these limitations, and so the greater type safety of monomorphic lambda expressions with explicit parameter types is preferred.

5.2 Using variables from the enclosing scope

An important design decision in creating a language extension for creating locally-defined functions is whether variables from the enclosing scope can be used in the body of the lambda expression, and if so, how they should be stored in the closure object. We have chosen to allow such uses, as many uses of lambda expressions (such as in the multiply_list() example above) require this feature. An alternative would be to completely disallow use of local variables, in which case no support for closures would be necessary, only local classes. We consider the loss in expressiveness severe enough to not pursue this alternative. Another alternative, suggested by Jeremy Siek, would be to require *all* local variables to be stored in the closure to be declared or listed explicitly using some syntax in the signature of the lambda function. We discuss this approach further in Section 5.3.

Given that local variables can be used transparently within the body of a lambda expression, another question is how to store them in the closure object. We allow changing the default storage mode from a **const** copy to storing a reference. Note that the actual implementation need not keep a reference to each particular local variable, if there is a way to keep a pointer to the stack frame or an equivalent instead.

Both ways of storing variables into closures, by copy and by reference, are problematic in their own ways. Storing local variables *by reference* leads to closure objects being second-class: unusable outside the function which created them. This would be a dangerous default, since one of the anticipated common uses of lambda expressions is callbacks. A lambda expression which does not refer to any non-copied local variables is usable in any program context.

Storing local variables *by copy* has other problems. For example, it is possible to create lambda functions which cannot be used outside their enclosing function, even if copies of the variables are stored in the closures. One example of this is if a local variable used (and stored by copy) in the closure is a pointer to another local variable. The pointee's lifetime is shorter than the lifetime of the closure, and so a dangling pointer if the closure escapes its enclosing function. References do not have this problem, because the object referred to is copied into the closure if the reference is used in the lambda expression's body. Slicing is also a possible problem, as a reference can refer to an object of a dynamic type derived from its static type; copying the referenced object using its static type would lead to only some of the actual object being copied. Copying of certain kinds of objects can also be very slow, which may not be expected to occur just from referring to a variable inside a lambda expression.

Note that some lambda functions will not compile with our chosen default. For example, streams are non-copyable types, and thus the following code is erroneous:

```
ofstream os("file.txt");
⟨⟩(int i) {os << i;}
```

The correct definition is:

```
ofstream os("file.txt");
⟨⟩(int i) extern(os) {os << i;}
```

Another kind of variable-like object that must be handled is the object ∗**this** inside a member function. We currently choose to copy the entire object, but storing it be reference is allowed with an optional declaration. The enclosing object has a weaker argument for being stored by copy than other local variables, but we copy it for consistency with the other design choices we have made.

5.3 Can we avoid all default behavior?

The major design issue with lambda expressions is how to handle references to variables defined outside of the body of the lambda expression. Both of the design choices discussed above (storing the variables either by copy or by reference) have their problems. Another viable design choice is to disallow references to local variables defined outside of the lambda function's body altogether, and allow the member variables of the closure to be explicitly declared and initialized as part of the definition of the lambda expression. In this approach, lambda expressions are very direct syntactic sugar for defining function objects. We demonstrate with an example, using invented syntax:

```
pair<int, int> p;
int s;
...
func(⟨⟩ (int x) → void
     : {int& sum = s; auto factor = 2 * p.first;}
     {sum += factor * x;});
```

Here, the block following the colon defines the member variables to be stored in the closure, and the expressions used to initialize them in the construction of the closure. Note that it would be possible to avoid explicitly defining the types of the member variables using **auto**.

The above definition would be translated to:

```
class __new_name {
  int& sum;
  int factor;
public:
  __new_name(int& sum, int factor) : sum(sum), factor(factor) {}
  void operator()(int x) {
    sum += factor * x;
  }
};
func(__new_name(s, 2 * p.first));
```

This design is possibly the one with the fewest surprises and the greatest flexibility. On the other hand, it provides relatively little benefit for programmers over explicitly defining local classes for their function objects, and we are uneasy with how verbose it can be.

5.4 The type of a closure

We have purposely left the types of closures unspecified to allow a variety of possible implementations. One possibility is to create a new local class for each lambda expression appearing in the program, as explained in more detail in Section 3. Each separate lambda expression would then have its own type, and so it would be statically known which lambda expression is being used at each call site. Thus, using a generic algorithm with a lambda expression in this model allows the body of the lambda expression to be inlined into the generic algorithm's body. It is possible to get rid of this exact static knowledge with library techniques in current C++, if it is not desirable for a particular situation (for example, when assigning one of several possible lambda expressions to a callback variable). A lambda expression can be used to initialize a tr1::function object, which involves a conversion that reduces the type information of the lambda expression to only its parameter and return types, giving more run-time flexibility but possibly losing the ability to inline calls to the expression. Another issue related to the types of closures is whether they are full classes, with the ability for users to create subtypes (derived classes) of them. This is probably not harmful, but the local variables in the closure are **private**, meaning that inheriting from a closure type has little benefit to the programmer.

Another design alternative would be to require that a lambda expression produces an object whose type is required to be an appropriate instance of tr1::function. This would allow functions to be easily stored in variables, and is similar to what many other languages with first-class functions use for their implementations. However, it would mean that a lambda expression would be similar to a function pointer in that the exact function called would not be expressed in its type; thus, generic algorithms would not be instantiated separately for each possible function used with them. This is likely to prevent compiler optimization in many cases.

5.5 Nested functions and inner classes

In our current design, all lambda expressions define anonymous functions of unspecified type. One possible change to this is to provide syntax for defining an explicitly named *nested function* (as provided by the GNU C Compiler). Named lambda functions can be simulated with our current proposal using a declaration such as

```
auto f = ⟨⟩(int x) → int {...};
```

A direct syntax may, however, also be useful. Direct syntax may also allow nested functions to be explicitly templated. Adding named nested functions would be a relatively small change to the proposal, and should not present any implementation difficulties.

Another possible addition is inner classes, as provided by Java. An inner class is similar to a normal class, except that the class's methods can directly access the function's local variables. Effectively, an inner class is similar to

a group of lambda expressions which share a set of local variables, but may also have other members. In C++, inner classes could, for example, be useful as multi-function callbacks, such as the visitors for algorithms in the Boost Graph Library. Inner classes can be implemented using a similar translation to local classes as is used for lambda expressions; local variable access would likely be implemented through hidden members in the translated inner classes, and so may present more difficulties than for single lambda functions. Also, exactly when an object of inner class type can be constructed is an important semantic issue, especially when many algorithms require that visitors used as their arguments are copyable in any context. More study is needed to assess how useful, and how implementable, inner classes are in the context of C++.

6 Alternative approaches

In addition to the lambda libraries for C++, several authors have designed limited forms of lambda or nested function functionality for C++. The closest to ours is by Oleg Kiselyov [Kis00], which uses a macro which expands to a local class and an instance of it, which is close to the translation in our proposal. However, his implementation does not allow local variables to be referenced in lambda expression bodies, and the operations that can be used on lambda expressions are very limited.

One article [Bre88] and proposal N0295 to the C++ committee [SH93] suggest adding nested functions to C++. Nested functions are similar to lambda expressions, but are defined as statements within a function body, and the resulting closure cannot be used unless that function is active. These proposals also do not include adding a new type for each lambda expression, but instead implementing them more like normal functions, including allowing a special kind of function pointer to refer to them. Both of these proposals predate the addition of templates to C++, and so do not mention the use of nested functions in combination with generic algorithms. Also, these proposals have no way to copy local variables into a closure, and so the nested functions they produce are completely unusable outside their enclosing function.

Herb Sutter suggests a way of simulating nested functions by changing the enclosing function into a class and its constructor, with the nested functions as member functions of that class [Sut99]. Multiple layers of nesting become quite difficult in this implementation, however, and the new classes created are not really functions. Our proposal, on the other hand, uses a language extension to create real closures even inside a normal function, and lambda expressions can be nested arbitrarily.

7 Conclusion

Anonymous functions within other functions, with access to local variables, are an important feature in many programming languages. They are also used in C++, as shown by the several libraries providing simulated lambda expression functionality [JP02, MS04, de 02]. This proposal presents a language extension to C++ to directly support this feature in a way that fits in with current practice, and with the rest of the language design. A suggested implementation strategy for this feature is also described. With this strategy, lambda expressions have the same performance characteristics as users expect from hand-created function objects. Thus, adopting this proposal as a part of C++0x will provide substantial convenience for users, with a reasonable implementation cost.

8 Acknowledgments

The authors wish to thank Jeremy Siek for suggesting the use of explicit declarations of which local variables will be used inside the lambda expression body and how to store them in the closure. Thanks also to Daniel Lidström for permission to use his post about Boost.Lambda as an example. Valentin Samko and Gabriel Dos Reis gave several detailed comments that notably improved the proposal. This work was supported by NSF grant EIA-0131354, a grant from the Lilly Endowment, and the first author was supported by a Department of Energy High Performance Computer Science Fellowship.

References

[ADH+98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams Iv, D. P. Friedman, E. Kohlbecker, Jr. G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[Bre88] Thomas M. Breuel. Lexical closures for C++. In *USENIX C++ Conference*, pages 293–304, October 1988. http://people.debian.org/~aaronl/Usenix88-lexic.pdf.

[Csh05] Microsoft Corporation. *C# Version 3.0 Specification*, September 2005. http://msdn.microsoft.com/vcsharp/future/default.aspx.

[de 02] Joel de Guzman. *Phoenix v1.2.1*. Boost, September 2002. http://www.boost.org/libs/spirit/phoenix/index.html.

[DGJP03] P. Dimov, D. Gregor, J. Järvi, and G. Powell. A proposal to add an enhanced binder to the library technical report. Technical Report N1455=03-0038, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1455.htm.

[ECM99] ECMA. *ECMAScript Language Specification*, 3rd edition, December 1999. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[Fou05] Python Software Foundation. Python 2.4.1 documentation. http://www.python.org/doc/2.4.1, 2005.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[GSW+05] Douglas Gregor, Jeremy Siek, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine. Concepts for C++0x (revision 1). Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.

[HAD04] Howard E. Hinnant, Dave Abrahams, and Peter Dimov. A proposal to add an rvalue reference to the C++ language. Technical Report N1690=04-0130, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming language C++, September 2004. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1690.html.

[JP02] Jaakko Järvi and Gary Powell. *The Boost Lambda Library*, 2002. www.boost.org/libs/lambda.

[JSD04] J. Järvi, B. Stroustrup, and G. Dos Reis. Decltype and auto (revision 4). Technical Report N1705=04-0145, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2004. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1705.pdf.

[Kis00] Oleg Kiselyov. Genuine lambda-abstractions in C++. http://okmij.org/ftp/c++-digest/#lambda-abstr, April 2000.

[MS04] Brian McNamara and Yannis Smaragdakis. Functional programming with the FC++ library. *Journal of Functional Programming*, 14(4):429–472, July 2004.

[PH+99] Simon Peyton Jones, John Hughes, et al. *Haskell 98: A Non-strict, Purely Functional Language*, February 1999. http://www.haskell.org/onlinereport/.

[SD05] Bjarne Stroustrup and Gabriel Dos Reis. A concept design (rev. 1). Technical Report N1782=05-0042, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2005.

[SH93] John Max Skaller and Fergus Henderson. A proposal for nested functions. Technical Report N0295=93-0088, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 1993. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1993/N0295.pdf.

[Str00] Bjarne Stroustrup. Binder problem and reference proposal (revised). Technical Report N1245=00-0022, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2000. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2000/n1245.ps.

[Sut99] Herb Sutter. GotW #58: Nested functions. http://www.gotw.ca/gotw/058.htm, July 1999.

[Wil03] Anthony Williams. Making local classes more useful. Technical Report N1427=03-0009, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, February 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1427.pdf.