# A proposal to add lambda functions to the C++ standard

**Valentin Samko**

*cxxpanel.valentin@samko.info*

**Project: ISO C++**

**Date: 2006-02-23**

**Document no: N1958=06-0028**

## 1. Introduction

A number of languages provide a way of passing code as arguments without having to define a separate named function [5]. For example, Algol 68 has downward funargs, Lisp has closures, in Smalltalk this is called “code blocks” which can be returned, passed as a parameter and executed later. Similar functionality is present in C# 2.0 (closures) and Java (anonymous classes). This concept is also not foreign to C++ as it extends the existing mechanisms and there are well known attempts (Boost.Lambda [9], Boost.Bind [8]) to introduce this functionality in C++ as a library solution.

*Closures typically appear in languages that allow functions to be first-class values, in other words, such languages allow functions to be passed as arguments, returned from function calls, bound to variable names, etc., just like simpler types such as strings and integers.*

*Wikipedia*

C++ has a concept of function objects which are the first class values, can be passed as arguments and returned, bound to variable names. Also, the well known Boost.Bind [8] library which was approved for TR1 was designed to bind parameters to function objects, creating new function objects. With the recent addition of normalised function pointers (**`tr1::function`**) this makes closures a missing logical extension of the C++ language, and in fact lambda functions ES017, ES062 [6] are in the list of active proposals of the C++ evolution group.

Many algorithms in the C++ Standard Library require the user to pass a predicate or any other functional object, and yet there is usually no simple way to construct such a predicate in place. Instead one is required to leave the current scope and declare a class outside of the function, breaking an important rule of declaring names as close as possible to the first use. This shows that lambda functions would add a great to the expressive power and ease of use of the C++ Standard Library.

We will refer to function objects which represent closures as lambda function objects, or lambda functions.

### 1.1. Motivation

- C++ Standard Library algorithms would be much more pleasant to use if C++ had support for lambdas. Lambda functions would let people use C++ Standard Library algorithms in many cases where currently it is easier to write a for loop. Many developers do not use function objects simply because of the syntactic overhead.
- A small set of trivial lambda functions can also be created with **`tr1::bind`**, but this proposal introduces a much better syntax which one can easily use (**`tr1::bind`** syntax is too complicated in many cases). For example, having

```
struct A { int foo(int y = 1, bool b = false); };
A a;
```

compare

```
tr1::function<int(int, int)> f = boost::bind(
 &A::foo,
 &a,
 tr1::bind(&A::foo, &a, 1, false),
 tr1::bind(&A::foo, &a, _1, false) <
 tr1::bind(&A::foo, &a, _2, false)
 );
```

with

```
tr1::function<int(int, int)> f = int(int x, int y) {
 return a.foo(a.foo(), a.foo(x) < a.foo(y));
};
```

- There is also a general understanding that closures are one of the most important missing C++ features, and there were many well known attempts to solve this problem on the library level (Boost.Bind [8], Boost.Lambda [9], Standard C++ library binders), but they introduce a very complicated and unreadable syntax, also leading to code which is very hard to debug and or analyse crash dumps. For example, in many environments developers are advised not to use these libraries as it takes significantly more time to analyse production problems one you have non trivial Boost.Bind or Boost.Lambda expressions in your call stack. Although the authors of these libraries did the best one can do in the current C++, and one can learn numerous highly non trivial programming techniques from these libraries, they introduce a new syntax for existing C++ constructs, making them very hard to read and understand. Another problem is that when one compiles source code which uses these libraries having inlining disabled, the performance is often severely affected. This again shows that lambda expressions are one of the most important missing features in C++ which can not be emulated in a reasonable way in a library.
- If one defines a function in a header file and needs to create a predicate for use in that function, that predicate class currently has to be defined outside of that function and so it is visible in every translation that includes that header file, although that predicate is supposed to be internal to that function.
- **`tr1::bind`** like solutions can not be used with many overloaded functions. For example, **`tr1::bind(&std::abs, _1)`** does not compile, and one has to write **`tr1::bind((Type(*)(Type))&std::abs, _1)`** where Type is the corresponding type name. Also, **`tr1::bind(&std::set<int>::find, _1, _2)`** does not compile for the same reason. This makes it very hard to use **`tr1::bind`** with the standard containers.
- Another problem with **`tr1::bind`** (although this can be considered to be an implementation detail, but the most popular implementation **`Boost.Bind`** has this problem) is that all the parameters are passed to the created function object by reference and thus **`void foo(int) {} boost::bind(&foo, _1) (1);`** fails to compile, since **`1`** is not a const object of type int, and you can not pass it as a non const int reference either.
- Most users will not use algorithms which require functional objects as long as one has to write more code to construct such function objects than one would write to “embed” that algorithm into their code. This effectively means that many generic programming patterns will not enter the mainstream until C++ supports a simple and efficient way to construct such function objects inline, just like one can construct other expressions. For example, having

```
typedef std::set<int> myset;
typedef std::vector<int> myvec;
```

the following four examples implement the same functionality in the function **`foo`** using lambda functions proposed in this document, using C++ algorithm with a custom predicate, using C++ algorithm with a predicate composed with **`tr1::bind`**, and with the algorithm code embedded in the function. Note, the last one is less optimal than any reasonable standard library implementation, and an implementation with **`tr1::bind`** is non portable as it assumes that **`myset::find`** does not have any additional parameters with default values.

(1) lambda function

```
void foo(
 myvec& v, const myset& s, int a) {
 // ...
 v.erase(
  std::remove_if(v.begin(), v.end(),
   bool(int x) {
     return std::abs(x) < a
       && s.find(x) != s.end(); }),
  v.end()
  );
}
```

(3) custom code instead of the standard algorithm

```
void foo(
 myvec& v, const myset& s, int a) {
 // ...
 myvec::iterator new_end = v.begin();
 for (myvec::iterator i = v.begin();
  i != v.end();
  ++i) {
  if ( ! (std::abs(*i) < a
    && s.find(*i) != s.end()) )
   *(new_end++) = *i;
  }
 v.erase(new_end, v.end());
}
```

(2) custom predicate

```
struct MyPredicate {
 MyPredicate(const myset& s, int a)
  : s_(s), a_(a) {}
 bool operator()(int x) const {
  return std::abs(x) < a_
    && s_.find(x) != s_.end();
 }
private:
 const myset& s_;
 int a_;
};

void foo(
 myvec& v, const myset& s, int a) {
 // ...
 v.erase(
  std::remove_if(v.begin(), v.end(),
   MyPredicate(s, a)),
  v.end()
  );
}
```

(4) `tr1::bind`

```
void foo(
 myvec& v, const myset& s, int a) {
 // ...
 v.erase(
  std::remove_if(
   v.begin(),
   v.end(),
   tr1::bind(
    std::logical_and<bool>(),
    (tr1::bind(
     (int(*)(int))&std::abs, _1) < a),
    (tr1::bind(
      (myset::const_iterator(myset::*)
(const int&)const)&myset::find,
       &s, _1) != s.end())
    )
   ),
  v.end()
  );
}
```

In addition to much more readable code (which example do you prefer to read to understand what **`foo`** does?), this example shows that with the current C++ standard the only sensible options are (2) and (3), and many developers pick (3) because it is shorter, and one does not need to define a separate structure which is visible to everyone else. Still, option (2) is generally faster than (3) as mentioned above. We note that if lambda functions are supported natively by the language, option (1) would be the most simple and brief as seen from this example. The size of the predicate functional object is also likely to be smaller with lambda functions as compiler may optimise away the extra reference and only store one pointer to the relevant stack frame in the predicate class.

- With lambda functionality described in this proposal we do not need a set of existing proposals, such as enhanced bindings [7], mem_fn adaptor [10], callable pointers to members [11].
- C++ algorithms are usually more optimal than a user implementation of the same functionality embedded into other functions, as the standard implementation of these algorithms may contain non trivial optimisations and be tailored for specific C++ standard library data structures. Because of the absence of a simple way to construct necessary predicates inline, many users miss these optimisations.
- One can not use **`tr1::bind`** in a portable manner with functions in the std namespace and member functions of standard C++ library containers as the implementation is allowed to add extra parameters with default arguments to these functions. There would be no such problem if C++ had a native support for lambda functions. The same problem exists with any 3rd party library when vendor adds a new argument with a

default value to a function which is used in **`tr1::bind`** expressions by a client.

- Recently Scott Meyers raised a question at comp.lang.c++.moderated (see the thread “Manual Loops vs STL Algorithms in the Real World”) asking whether it's really reasonable to expect C++ programmers to use STL algorithms instead of writing their own loops. It was stated that the real life code is usually less trivial than the common examples of using standard library binders, and function objects overcomplicate the code. This problem would be solved by lambda functions proposed in this document as lambda functions introduce a very short and convenient notation to define function objects inline and pass them to any algorithms.
- A short but a very important note in favour of native support for lambda functions is that with lambda functions the overall quality of C++ code would improve with the new code containing less bugs which very often occur when programmers “embed” algorithms into their code. I.e. lambda functions would give a much needed boost to the reuse of generic algorithms.

## 1.2. Required functionality

- Lambda functions must be the first class C++ citizens, i.e. objects. These objects must have class types, and these classes must have the "result_type, arg1_type, ..." typedefs and the corresponding operator().
- Lambda objects must be copyable. When a lambda object is copied, all the objects and references bound to that lambda object must be copied.
- A lambda must have the same access to the names and entities outside of the lambda definition as the scope enclosing that lambda definition. There is already a similar case for local classes (9.8/1) where declarations in a local class can use type names, enum values, extern variables and static and global variables from the enclosing scope.
- Lambdas must be significantly easier to read, write and debug than any possible library solutions.
- Lambdas must be compatible with the proposed **`tr1::bind`** and **`tr1::function`** libraries, i.e. one should be able to create **`tr1::function`** objects from lambda objects and bind parameters to lambda objects with **`tr1::bind`**.

## 1.3. Definitions

**Lambda object** – A function object of an implementation dependent class type with operator() and result_type, etc. typedefs which can be created by a primary expression.

**Declaration of a lambda object** – A primary expression which creates a temporary lambda function object.

**Definition of a lambda object** – A declaration of a primary object is also a definition.

**Lambda body** – A code block which can be executed, given a set of parameters and context.

**Local lambda** – Lambda definition in a function, or in a lambda body.

**Lambda** – An expression which contains code and refers to certain variables, which is not evaluated immediately, and can be passed to functions which will evaluate it when needed, or stored to be evaluated in the future.

**Lambda bound values initialiser** – Declaration and definition of variables which are contained in a lambda object, copy constructed when the lambda object is constructed or copy constructed. They can be used to bind any values to the lambda function by value.

## 1.4. Lambda objects and normalised function pointer type

1. We define normalised function pointers as normalised types for any expressions which can be called with a function call syntax. A main requirement for such normalised function pointers is that all the normalised function pointers which return objects of the same type and accept parameters of the same type do have the

same type and can be copied and assigned and it does not matter whether they refer to plain function pointers, or to complicated function objects. **`tr1::function`** is an example of a library solution for such normalised function pointers. Some languages support normalised function pointers directly.

2. Lambda functions are not required to be normalised function pointers. Lambda objects created by different code are not required to be assignable or to have the same layout, even if they return values of the same type and their arguments have same types. One can always normalise any lambda object using the library solutions like **`tr1::function`**. Even if normalised function pointers are added to the C++ standard, they will not affect lambda functions in any way since lambda functions are orthogonal to normalised function pointers (in the same way as the result of a **`tr1::bind`** expression is orthogonal to **`tr1::function`**).
3. The fact that lambda objects are function objects and not function pointers is consistent with existing C++ practices as we already have a notion of function objects, which are objects classes with operator(). Also, normalised function pointers would have the same limitation that any other function pointers have, for example function calls through such pointers can rarely be inlined.
4. Unlike function pointers, lambda objects are not comparable with 0 and do not have operator ! as they are always valid, one can not have an uninitialised lambda object.

### 1.5. Proposed syntax

```
  ret_type(type1 value1, type2 value2, ...) { statements }
```

All the names and entities visible and accessible in the scope where lambda is declared must be also visible and accessible in the body of the lambda object. For example

```
void f(int x) {
 std::vector<int> v;
 // ...
 std::remove_if(v.begin(), v.end(), void (int& n) { n < x; });
}
```

Optionally, one can bind values to lambda objects which will have the same life time as the lambda object by using the lambda bound values initialiser, such as **`(typea boundvalue1(x), typeb boundvalue2(y), ...)`.** This initialiser can be used to pass variables to the lambda by value, as by default nothing is passed to the lambda, all the variables in the enclosing scope are automatically visible in the lambda function body and it is undefined behaviour if lambda refers to variables whose storage has been released. The complete lambda definition with bound values initialiser is

```
ret_type (type1 value1, type2 value2, ...)
 : (typea boundvalue1(x), typeb boundvalue2(y), ...)
{ statements }
```

For example

```
void set_callback(tr1::function<bool(int)>);
void foo(int t) {
 set_callback(bool(int i) : (int number(t)) { std::cout<<(i + number); });
}
```

A lambda definition is a primary expression and so it can be used anywhere where any other primary expression can be used. For example,

```
int x = 0;
struct A {
 tr1::function<void()> f;
 A() : f(void(){ ++x; }) {}
 void foo(tr1::function<bool()> p = bool(){return x<0;}) { p(); }
 static tr1::function<int(bool)> f2;
```

```
 void bar() { throw void() { --x; }; } // can only be caught in (...)
};
tr1::function<int(bool)> A::f2 = int(bool b) { return b ? 1 : 2; };
```

## 2. Specification

### 2.1. Lambda definition

A lambda definition defines an implementation dependant lambda class with external linkage and a temporary object of that class (12.2). Lambda definitions are primary expressions and have the following grammar

```
lambda-definition:
  type-specifier lambda-declarator lambda-bound-values-declarator_opt { lambda-body }

lambda-declarator:
  ( parameter-declaration-clause )

lambda-bound-values-declarator:
  : ( lambda-bound-values-list )

lambda-body:
  compound-statement

lambda-bound-values-list:
  lambda-bound-value
  lambda-bound-values-list, lambda-bound-value

lambda-bound-value:
  decl-specifier-seq declarator(assignment-expression)
  decl-specifier-seq abstract-declarator(assignment-expression)
```

where parameters in the *parameter-declaration-clause* are not visible in *assignment-expression* elements in the *lambda-bound-values-declarator*.

### 2.2. Lambda classes

1. Lambda objects have class types, and expressions
   - **`ret_type (type1 param1, type2 param2, ...) { statements }`**
   - **`ret_type (type1 param1, type2 param2, ...) : (type3 boundvalue1(v1), type4 boundvalue2(v2), ...) { statements }`**

   create temporary lambda objects of classes with implementation dependent names, where each such expression can possibly result in a different class. The fact that the type of lambda classes is unspecified is consistent with the proposal for an enhanced binder [7] which was approved for TR1, where the type of the returned function objects is also unspecified.
2. If a lambda is defined within a function template, or a member function of a class template, it is essential that every unique instantiation of the template yields a unique type for the lambda class, just as every unique instantiation of the template yields a unique address for the function in which the local class is defined. If a local class is defined within a function which is itself a template, or is a member of a template, then instantiations of the template with the same set of template parameters must yield the same instance of the local class, even if the instantiations are performed in different translation units.
3. Lambda classes may have any implementation dependent names, and classes representing different lambda objects are allowed to have different names even if they return values of the same type and have the same set of parameters. Implementation must guarantee that lambda class names do not clash with any user defined names and the observable behaviour does not depend on names of lambda classes.
4. Lambda classes must have the public "result_type, arg1_type, ..." typedefs and public operator() const without

exception specification. When operator() is called, it must invoke the lambda body. Also, no “this” pointer is declared in the lambda body, but if another “this” is visible in the scope where the lambda is declared, that “this” is visible in the lambda body, the same stands for the operator(), i.e. the lambda body can not recursively call itself directly.

5. The implementation is required to define a public copy constructor, and not to define operator =.
6. The implementation is required to define a public default constructor if and only if the body of the corresponding lambda definition does not refer to anything except global, static and extern variables and enums and does not contain any bound values.
7. **`sizeof(lambda_type)`** is implementation dependent and is allowed to be different for lambda classes defined by different lambda definitions.
8. Implementation is allowed to add any member variables to the lambda class.
9. Declarations in the lambda body can use any type names, variables, functions and enumerators from the enclosing scope.
10. Types of all the lambda parameters and bound values must have external linkage.

## 2.3. Lambda behaviour

We follow the approach used to introduce unnamed namespaces (7.3.1.1) and define lambda behaviour in terms of the behaviour of existing language constructs in the current standard.

Lambda definition behaves as if

1. Lambda definition is replaced by a simple type specifier of a class followed by a parenthesized expression list (5.2.3) where that class has a globally unique name is defined prior to the declaration of the function the lambda definition is defined in.
2. All the enclosing scope variables visible at the point of lambda definition which names are used in the lambda body are passed to the constructor of that class, as well as all the bound values.
3. Constructor of that class accepts the local scope variables by reference, and bound values by types specified in the lambda bound values initialiser.
4. That class has members variables with types and names specified in the lambda bound values initialiser, and they all are initialised in the member initialiser of the class constructor.
5. For every local scope variable accessed from the lambda body, that class has a member variable which has the type of a reference to the original variable, or of a reference to the type that variable refers to if that variable is a reference itself. All these references are initialised in the initializer list of the class constructor.

For example, having

```
std::string s = “test”;
```

the observable behaviour of

```
tr1::function<int(std::string, bool)> f =
 int(std::string x, bool b) { std::cout << s << x ; } ;
```

the same as of

```
class unique_class_name {
public:
 typedef result_type int;
 typedef arg1_type std::string;
 typedef arg2_type bool;
 unique_class_name(std::string& s_) : s(s_) {}
 int operator()(std::string x, bool b) const { std::cout << s << x; }
private:
 std::string& s;
```

```
};
tr1::function<int(std::string, bool)> f = unique_class_name(s);
```

and having

```
bool b = false;
std::string t = “test”;
```

the observable behaviour of

```
tr1::function<double(int, const std::string&)> f =
 double(int i, const std::string& s)
  : (bool f(b), std::string p(“test”))
 { std::cout << s << p << f << i << t; }
```

is the same as of

```
class unique_class_name {
public:
 unique_class_name(std::string t_)
  : t(t_), f(b), p(“test”) {}
 typedef result_type int;
 typedef arg1_type std::string;
 typedef arg2_type bool;
 double operator()(int i, const std::string& s) const
 { std::cout << s << p << f << i << t; }
private:
 bool f;
 std::string p;
 std::string& t;
};
tr1::function<double(int, const std::string&)> f = unique_class_name(b, t);
```

## 2.4. Linkage

1. Lambda classes should have the linkage of the enclosing function (or of the class if the lambda is defined in a class constructor or destructor, or in the member initialiser in the constructor), or of the variable being initialised if lambda class is defined in a *simple-declaration* in the namespace scope.
2. If a lambda is defined in a function, all the names defined in that function are visible in the that class with the globally unique name.

## 2.5. Access to names and entities from inside the lambda body

1. The lambda body and the lambda bound values initialiser must have the same access to the names outside of the lambda definition as the scope enclosing that lambda definition.
2. All the entities visible in the lambda body may actually be references to the original variables, but this must not be observable in the lambda body.
3. It is legal for lambda objects to refer to local references even if lambda objects outlive these references but not the referenced objects.
4. Name visibility in a lambda should be the same as if one had a function object class (in a unique namespace) with external linkage, which had members with the same names and types (references) as all the variables defined in the local scope and accessible from the point where lambda was declared, including function parameters. Whether types of these "member variables" are references or not is implementation dependent.
5. It is undefined behaviour if there is a valid lambda object which body refers to a name of an object which storage is released.

## 2.6. Where lambda expressions can be used

Anywhere where a primary expressions may be used. For example,

1. In function bodies (*function-body*) of functions
2. In an *assignment-expression* of function or operator *parameter-declaration-clause*
3. In variable declarations (*simple-declaration*) in global and namespace scopes.

(1) and (2) are implementable for functions with external linkage. In case of functions with internal linkage, this may be done, as this is implemented in VC++ compilers (v7, v8). In case of (3), if the variable being declared has internal linkage or is in an unnamed namespace, the implementation must create a different lambda-object in every translation unit. For example,

```
int x;
namespace { int y; }
tr1::function<void()> f = void() { x += y; };
```

is equivalent to

```
namespace { lambda_class_name lambda_object; }
tr1::function<void()> f = lambda_object;
```

This would also work with proposal for type deduction N1894 [1]. For example one could write

```
int x;
namespace { int y; }
auto f = void() { x += y };
```

This approach would also result in correct code if a lambda object is used to initialise a variable in an unnamed namespace. For example

```
int x;
namespace { int y; }
namespace { tr1::function<void()> f = void() { x += y; }; }
```

## 2.7. Impact on existing code

This proposal is only an extension and there is no impact on existing code.

## 2.8. Examples

```
template<class T> void inherit(T t) {
  struct X : public T {
    X() {} // ill-formed, classes generated by lambda definitions which refer
to local variables do not have default constructors
    void foo() {
      --k; // ill-formed, 'k' is is not declared here, it is only declared in
the lambda function body
    }
  };
  t(); // valid
  T t2(t); // valid
}

void foo() {
  int k = 0;
  inherit(void () { ++k; });
}
```

```
class B : public tr1::enable_shared_from_this<B> {
```

```
  int n;
  B() {}
  B(const B& b) : n(b.n) {}
public:
  static tr1::shared_ptr<B> create() {
    return tr1::shared_ptr<B>(new B());
  }
  tr1::function<void(int&)> callback() const {
    return void(int& x)
      : (tr1::shared_ptr<const B> me(shared_from_this()))
    { x += n; };
  }
  // Note, this is not a misprint, x+=n is correct as well as x+=me->n as
this lambda object already contains a shared pointer to this object, so it is
not destroyed until the last copy of this lambda object is destroyed.
  // This callback can still be used even if user does not have any direct
shared pointers to this object, as this lambda object will get a copy of a
shared pointer to this object.
};
```

```
namespace {
 int x = 0;
 tr1::function<void()> f = void() { ++x; };

 tr1::function<tr1::function<void()>()> f2 =
  tr1::function<void()>() { return void() { ++x; }; };

 typedef tr1::function<int(int)> integer_transform;
  typedef tr1::function<integer_transform(integer_transform)
  > function_operator;

 function_operator fop = integer_transform(integer_transform t) {
  return int (int x) : (integer_transform original_t(t)) {
   return original_t(x) % 100;
  };
 };
 // this creates a function object transformer fop which can be used to
transform any function object which converts integer to another integer into
another similar function object, which executes the original function object
and returns the remainder after division of the result by 100.
 integer_transform t1 = int(int x) { return x*x; };
 integer_transform t2 = fop(t1);
}
void foo() {
 f();
 f2()();
 t2(10); // equivalent to fop(t1)(10)
 fop(t1)(10);
}
```

The following would be possible if the proposal [12] is accepted.

```
std::set<
  int, decltype(bool (int x, int y) { return std::abs(x) < std::abs(y); })
  > s;
struct A { std::string name; };
std::set<
  A, decltype(bool (const A& x, const A& y) { return x.name < y.name; })
  > sa;
```

## 3. Protection against accidental misuse

1. Lambda object may outlive local variables used by that lambda object and once these variables are destroyed and their storage is released, even an existence of a lambda function which refers to them results in undefined behaviour.
2. It a general case, is not possible for a compiler to determine at compile time that lambda will not outlive any variables it refers to.
3. It is undefined behaviour if there is a lambda object which body refers to a name of an object which storage is released, see 2.5/6 in this document. For example,

```
tr1::function<void()> foo() {
 int c = 0;
 return void(){ ++c; };
}
```

   and

```
bool b = true;
tr1::function<void()> foo() { int x=0; return void() { if (b) ++x; }; }
void bar() { b = false; foo()(); }
```

   results in undefined behaviour.

4. A case when a lambda object still exists which refers to local variables after they are destroyed is similar to existing C++ practices, for example

```
struct A { A(int& i) : i_(i) {} int& i_; };
A f() {
  int j=0;
  A a(j);
  // ...
  return a;
}
```

   will lead to similar undefined behaviour and no compiler diagnostic is required. This shows that our case is consistent with the existing C++ practices.

### 3.1. Run time error detection

Most of such problems may be detected at run time by the implementation (many compilers already have options to add buffer overrun and other run time error detection). For example, if a lambda is defined in a function and its body refers to an object defined with automatic storage in that function, then implementation may store a counter of all the lambda objects of that type created by that function call or by copy construction, and if that counter is non zero when storage for local variables in that function is released, then the error may be reported at run time.

So, for

```
void foo() {
  int a = 0;
  bar(void() { ++a; });
}
```

The implementation will produce the equivalent of

```
struct lambda_class {
  lambda_class(size_t& counter, int& a) : counter_(counter), a_(a) {}
  lambda_class(const lambda_class& x) : counter_(x.counter_), a_(x.a_)
{ ++counter_; }
  ~lambda_class() { --counter_; }
  int& a_;
  size_t& counter_;
};
```

```
tr1::function<void()> foo() {
  int a = 0;
  struct guard {
    guard() : counter(0) {}
    size_t counter;
    ~guard() { if (counter) report_error(); }
  } g;
  bar(lambda_class(g.counter, a));
}
```

which would call implementation internal function **`report_error()`** if function **`bar`** stores a copy of the passed lambda object.

This run time error detection is possible for variables with automatic storage defined in the function (or another lambda body) where a lambda was defined. Similar run time checks for member variables (when lambda is defined in a member function) or for dynamically allocated objects are much harder to implement and will result in significant performance and memory usage overhead. For example,

```
struct A {
 int member_int;
 tr1::function<void()> foo() { return void() { ++member_int; }; }
};
tr1::function<void()> bar() {
 A a;
 return a.foo();
}
```

will result in undefined behaviour but the error will not be reported at run time. We also note that when creating function objects with **`tr1::bind`** for member functions, there are no run time checks whether the object which member function is being called was not destroyed yet.

There is no need for the standard to require implementations to detect these errors at run time, as such run time error detection can be optionally provided by an implementation and would be conforming with the standard (as the error is only detected when an undefined behaviour would occur otherwise). This would be consistent with the spirit of C++, as similar run time checks are not required when the program uses a reference to an object which storage was released. For example,

```
struct X { X(const int& x) : v(x) {} const int& v; };
X foo(int t = 1) { return X(t); }
int y = foo().v;
```

results in undefined behaviour, but no run time error detection is required and it is up to the implementation to detect such errors at run time.

### 3.2. Garbage collection

Another approach to avoid problem with lambda objects accessing variables which no longer exist requires garbage collection for all the variables (even primitives on the stack), and this is not achievable in C++. Any workarounds would only cover a few useful cases (i.e. never 100% coverage) and it would be against the spirit of C++. Also, such workarounds may introduce performance loss and possible memory allocation errors in the most simple cases (for example, allocation of local variables on the heap if they are used in a closure, as this is done by C#). Also we need to note that even if GC is introduced in C++, this still will not solve the general problem with local primitives on the stack. For example there is a similar problem in Java, where closures can not refer to non final local variables.

Therefore, the problems associated with lambda functions in the absence of GC should not stop the introduction of the lambda functions in C++, as is it very unlikely that such a GC (even for local variables on the heap) will be introduced in C++ in a foreseeable future.

### 3.3. Copying local scope variables to lambda by default

If objects with automatic storage defined in functions are copied to the lambda object by default (and not referenced as this is described in this proposal) the problem with lambda objects outliving the function scope would be less of an issue. Unfortunately, this approach has other very significant problems.

1. Copying would invalidate iterators if both, container and iterator to that container are copied.
2. Copying will result in slicing.
3. Copying will lead to unexpected results with pointers and references, as pointers copied from the local scope will still point to the original objects and not to the copied ones.
4. Copying objects of non trivial types will result in performance problems.
5. Copying will make access to the function scope variables inconsistent with access to the namespace scope variables, as the second will not be copied.

## 4. Required changes to the standard

1. 1.8/1

   An object is created by a *definition* (3.1), by a *new-expression* (5.3.4) **[Add** by a lambda definition (8.6)**]** or by the implementation (12.2) when needed.
2. **[Add:** 3.3.2/5 – Names declared in lambda definitions are local to these lambda definitions. **]**
3. **[Add:**

   3.3.4/8 – Lambda scope [basic.scope.lambda]

   1. The potential scope of a name declared in a lambda definition consists of the declarative region following the name's point of declaration and of the lambda function body.
   2. A name declared in the lambda definition hides a declaration of the same name in the enclosing scope, which scope extends to the lambda function body. **]**
4. 3.9.2 Compound types

   -- classes containing a sequence of objects of various types (clause 9), a set of types, enumerations and functions for manipulating these objects (9.3), and a set of restrictions on the access to these entities (clause 11) **[Add** or defined by a lambda definition**]**.
5. 5.1/3 The keyword *this* shall be used only inside a non-static class member function body (9.3) **[Remove:** or**]** in a constructor mem-initializer (12.6.2) **[Add** or in lambda definition defined in a non static class member function body**.** When the keyword *this* is used in a lambda definition, it refers to the pointer to the enclosing function's class.**]**
6. 7/1 Declarations [dcl.dcl]

   declaration:

   **[Add** *lambda-function-definition* **]**
7. 7/2

   *A declaration that declares a* function or defines a class, namespace, template**, [Add** lambda function **]** or function also has one or more scopes nested within it.
8. **[Add**

   8.6 “Lambda definition”

   Lambda definition defines an implementation dependant lambda class with external linkage and a temporary object of that class (12.2). Lambda definitions are primary expressions and have the following grammar

```
lambda-definition:
    type-specifier lambda-declarator lambda-bound-values-declarator_opt {lambda-body}
lambda-declarator:
    ( parameter-declaration-clause )
lambda-bound-values-declarator:
    : ( lambda-bound-values-list )
lambda-body:
    compound-statement
lambda-bound-values-list:
    lambda-bound-value
    lambda-bound-values-list, lambda-bound-value
lambda-bound-value:
    decl-specifier-seq declarator ( assignment-expression )
    decl-specifier-seq abstract-declarator ( assignment-expression )
```

   1. As lambda-function-definition defines a class and a variable, it can be defined in any scope where a temporary is acceptable.
   2. Lambda objects must be copyable. When a lambda object is copied, all the objects and references bound to this lambda objects must be copied.
   3. Lambda objects are temporary objects, created lambda definitions.
   4. As lambda objects are temporary objects, they are destroyed as the last step in evaluating the full-expression (1.9) that (lexically) contains the point where they were created.

   **[ The contents of sections 2.2 – 2.5 of this proposal should be inserted here ] ]**
9. **[Add** 9/6 – Classes can also be defined by lambda function definitions (8.6) **]**
10. 12.1 Constructors [class.ctor]

    ... An implicitly-declared default constructor for a class is *implicitly defined* when it is used to create an object of its class type (1.8). **[Add: If it is used to create an object of a class generated by a lambda definition which refers to anything but static, global and external variables and enum values or has bound variables, the program is ill-formed.]**
11. 12.2/1 Temporary objects [class.temporary]

    Temporaries of class type are created in various contexts: binding an rvalue to a reference (8.5.3), returning an rvalue (6.6.3), a conversion that creates an rvalue (4.1, 5.2.9, 5.2.11, 5.4), throwing an exception (15.1), entering a handler (15.3), **[Add** lambda definition (8.6) **]** and in some initializations (8.5).”
12. A.4

```
primary-expression:
    literal
    this
    ( expression )
    id-expression
[Add: lambda-expression ]
[Add
    lambda-expression
        lambda-function-definition ]
```

13. A.6

```
    declaration:
        [Add lambda-definition ]
```

## 5. Implementation notes

1. Local functions which have access to the local variables are quite similar to the lambda functions and are

supported by the GNU C compiler. It implements taking the address of a nested function using a technique called trampolines. The downside of this approach is that it requires executable stack. The upside is that there is implementation experience of a technique similar to what we are proposing.

2. Once we have local classes with external linkage [2], lambdas are much easier to implement.
3. Many implementation details including generation of the unique external names and to the linkage, relevant to this proposal and explaining how such a functionality may be implemented are explained in detail in [2].
4. Lambda functions may lead to a more optimal code than with wrapper-helper classes defined by user. For example, consider

```
struct helper_functor {
 helper_functor(int& a, std::string& b, int& c, double& d)
  : local_a(a), local_b(b), local_c(c), local_d(d)
 int& local_a;
 std::string& local_b;
 int& local_c;
 double& local_d;
 bool operator()(const MyClass& v) const {
  return v.foo(local_a, local_b) ? v.bar(local_c, local_d) : true
 }
};

void foofoo(int a, std::string b) {
 std::vector<MyClass> v = get_data();
 int c = barbar();
 double d = 3.14;
 std::remove_if(v.begin(), v.end(), helper_functor(a, b, c, d));
}
```

Here **`sizeof(helper_functor)`** is at least the size of 4 references, and this object will be copied by the for_each algorithm, possibly many times. Now, consider the equivalent with C++ lambdas

```
void foofoo(int a, std::string b) {
 std::vector<MyClass> v = get_data();
 int c = barbar();
 double d = 3.14;
 std::remove_if(
  v.begin(), v.end(),
  bool (const MyClass& v) { return v.foo(a, b) ? v.bar(c, d) : true; });
}
```

In addition to having simpler and easier to understand source code, this version may be more optimal as implementation is not required to store in the lambda object references to all the variables accessed in the lambda definition body. For example, implementation may only store one pointer to the stack frame, thus reducing the size of the lambda object, and that will result in a more optimal code.

## Annex A – Notes

1. Although this proposal does not provide any means to pass lambda function objects to functions which require a function pointer parameter, this functionality can be added in the future for lambdas which do not refer to any local variables and do not have any bound variables by adding a conversion to function pointer operator to the corresponding lambda class.
2. The lambda definition syntax in this proposal does not support lambda classes with template operator(). As this functionality can be added later and it will not conflict with anything in this proposal (apart from

argX_type typedefs) this is left out of this proposal for now.

3. If the proposal for decltype and auto keywords [12] is accepted then return_type and argX_type typedefs will not be required as one will always be able to deduce the return type and argument types from the operator(). This will also simplify introduction of lambdas with template parameters, as such lambda classes can not have argX_type typedefs for template parameters.

## Acknowledgements

Thanks to everyone who attended my presentation on lambda functions, commented and helped with advices during the presentation and by email, especially Thorsten Ottosen, Anthony Williams, Roger Orr, Lois Goldthwaite, Francis Glassborow, Tim Penheney, Clarles Reindorf, Sam Saariste, Jonathan Wakely.

## References

[1] Deducing the type of variable from its initializer expression http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1894.pdf

[2] Making Local Classes more Useful - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1427.pdf

[3] C++ lambda preprocessor - http://home.clara.net/raoulgough/clamp/index.html

[4] Thoughts about the possibility to add lambda to C++ - http://www.msobczak.com/prog/articles/lambdacpp.html

[5] Wikipedia definition of Closure http://en.wikipedia.org/wiki/Closure_(computer_science)=

[6] C++ Evolution Working Group -- Active Proposals http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1650.html

[7] A Proposal to Add an Enhanced Binder to the Library Technical Report http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1455.htm

[8] Boost.Bind C++ library http://www.boost.org/libs/bind/bind.html

[9] Boost.Lambda C++ library http://www.boost.org/doc/html/lambda.html

[10] A Proposal to Add an Enhanced Member Pointer Adaptor http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1432.htm

[11] A proposal to make pointers to members callable http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1695.html

[12] Decltype and Auto proposal - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1705.pdf

[13] GCC nested functions - http://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/Nested-Functions.html#Nested-Functions

[14] Local closures for C++ - http://people.debian.org/~aaronl/Usenix88-lexic.pdf.