Lambda expressions and closures for C++ (Revision 1)

Document no: N2329=07-0189

Jaakko Järvi*
Texas A&M University

John Freeman
Texas A&M University

Lawrence Crowl
Google Inc.

2007-06-24

1 Introduction

This document proposes *lambda expressions* to be adopted to C++. The proposal is a major revision of the document N1968 [WJG$^+$06], and draws also from the document N1958 [Sam06] by Samko. The differences to N1968 are:

- Most of the discussion of background and motivation for lambda expressions is omitted.
- There are some changes in the proposed syntax.
- Lambda functions can be defined generically: their parameter types need not be explicitly specified.
- Behavior with constrained templates (concepts) is explained in detail.
- The proposed semantics have changed to *not* allow referring to variables defined in outer scopes in the body of a lambda expression. Instead, the programmer explicitly declares the variables to be stored, and how they are stored, in the closure.

The proposed standard wording is not yet included.

We use the following terminology in this document:

- *Lambda expression* or *lambda function*: an expression that specifies an anonymous function object
- *Closure*: An anonymous function object that is created automatically by the compiler as the result of evaluating a lambda expression. Closures consists of the code of the body of the lambda function and the *environment* in which the lambda function is defined. In practice this means that variables referred to in the body of the lambda function are stored as member variables of the anonymous function object, or that a pointer to the frame where the lambda function was created is stored in the function object.

The proposal relies on several future additions to C++, some of which are already in the working draft of the standard, others likely candidates to be included into the standard. These include rvalue references [Hin06], the **decltype** [JSR06a] operator, and the ability to deduce the type of a variable from its initializer expression using **auto** [JSR06b].

2 In a nutshell

The use of function objects as higher-order functions is commonplace in calls to standard algorithms. In the following example, we find the first employee within a given salary range:

*jarvi@cs.tamu.edu

```
class between {
  double low, high;
public:
  between(double l, double u) : low(l), high(u) { }
  bool operator()(const employee& e) {
    return e.salary() >= low && e.salary() < high;
  }
}
...
double min_salary;
...
std::find_if(employees.begin(), employees.end(),
              between(min_salary, 1.1*min_salary));
```

The constructor call between(min_salary, 1.1∗min_salary) creates a function object, which is comparable to a *closure*, a term commonly used in the context of functional programming languages. A closure stores the *environment*, that is the values of the local variables, in which a function is defined. Here, the environment stored in the between function object are the values low and high, which are computed from the value of the local variable min_salary. Thus, we store the information necessary to evaluate the body of the **operator**()() in the closure.

The syntactic requirement of defining a class with its member variables, function call operator, and constructor, and then constructing an object of that type is very verbose, and not well-suited for creating function objects "on the fly" to be used only once. The essence of this proposal is a concise syntax for defining such function objects—indeed, we define the semantics of lambda expressions via translation to function objects. With the proposed features, the above example becomes:

```
double min_salary = ...
...
std::find_if(employees.begin(), employees.end(),
             <>(e : auto l = min_salary, auto h = 1.1*min_salary) { e.salary() >= l && e.salary() < h });
```

We propose syntactic shortcuts, described later, that considerably shorten the lambda function above.

3 Proposal

Lambda expressions are *primary-expressions*. Lambda functions are defined with the following syntax:

lambda-expression:
  lambda-head exception-specification$_{opt}$ *lambda-body*

lambda-head:
  <>(*lambda-parameter-clause*$_{opt}$ *lambda-local-var-clause*$_{opt}$)
  <&>(*lambda-parameter-clause*$_{opt}$)

lambda-body:
  { *expression* }
  −>*type-id* { *expression* }
  −>*type-id compound-statement*

lambda-parameter-clause:
  lambda-parameter
  lambda-parameter , *lambda-parameter-clause*

lambda-parameter:
  identifier
  identifier = *assignment-expression*

  decl-specifier-seq declarator
  decl-specifier-seq declarator = *assignment-expression*

lambda-local-var-clause:
  : *lambda-local-var-list*$_{opt}$

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

lambda-local-var:
  decl-specifier-seq declarator = *assignment-expression*
  &$_{opt}$ *identifier*
  this

In the following, we introduce the proposed feature informally using examples of increasing complexity. We will need the helper function make, below, that can be used to write an expression of an arbitrary type T, as make<T>():

```
template <class T> T make();
```

3.1 Lambda functions with no external references

We first discuss lambda functions that have no references to variables defined outside of its parameter list. We demonstrate with a binary lambda function that invokes **operator**+ on its arguments. The most concise way to define that lambda function is as follows:

```
<>(x, y) { x + y }
```

No types for the parameters need to be defined. This kind of a lambda function is *generic*. The types for the arguments will be deduced similarly to template argument deduction. This is discussed in Section 6.

It is also possible to define parameter types explicitly:

```
<>(int x, int y) { x + y }
```

This lambda function can only be called with arguments that are of type **int**, or convertible to type **int**. We refer to a lambda function that specifies all of its argument types as *non-generic*.

In both of the above examples, the body of the lambda function is a single expression. The return type does not have to be specified; it is deduced to be the type of the expression comprising the body. Return type deduction is defined with the help of the **decltype** operator. Above, the return type is defined as **decltype**(x + y).

Explicitly stating the return type is allowed as well, for which the syntax is as follows:

```
<>(int x, int y) ->int { x + y }
```

It is possible to define lambda functions where the body is a block of statements, rather than a single expression. In that case, the return type must be specified explicitly:

```
<>(int x, int y) ->int { int z; z = x + y; return z; }
```

The rationale for requiring the return type to be specified explicitly is two-fold. First, the body of a lambda function could contain more than one return statement, and the types of the expressions in those return statements could differ. Such definitions would likely have to be flagged out as ambiguous. Second, implementing return type deduction from a statement block may be complicated. That is, the return type is no longer dependent on a single expression, but rather requires analyzing a series of statements, possibly including variable declarations etc.

The semantics of lambda functions are defined by translation to function objects. For example, the last lambda function above is defined to be behaviorally equivalent with the function object below. The proposed translation is somewhat more involved; we describe the full translation in Section 5.

```
class F {
public:
  F() {}
  int operator()(int x, int y) const {
    int z; z = x + y; return z;
  }
};
F() // create the closure
```

We summarize the rules this far:

- Each parameter slot in the parameter list of a lambda function must either consist of an identifier, or of an identifier followed by a default argument, or of an "ordinary" function parameter declaration with a non-abstract declarator. A single identifier is thus interpreted as the name of the parameter, not a type. If no type is given for a particular parameter, that parameter is generic and its type is deduced from the use of the lambda (see Section 6).
- The body of the lambda function can either be a single expression (enclosed in braces) or a block.
- If the body of the lambda function is a block, the return type of the lambda function must be explicitly specified.
- If the body of the lambda function is a single expression, the return type of the lambda function may be explicitly specified. If it is not specified, it is defined as **decltype**(e), where e is the body of the lambda expression.

3.2 External references in lambda function bodies

References to local variables declared outside of the lambda function bodies have been the topic of much debate. Any local variable referenced in a lambda function body must somehow be stored in the resulting closure. The earlier proposal [WJG+06] called for storing such local variables by copy and required an explicit declaration to instruct that a variable should be stored by reference instead. Another proposal by Samko [Sam06] suggested the opposite. However, neither alternative gained wide support as both approaches have notable safety problems. By-reference can lead to dangling references, by-copy to unintentional slicing of objects, expensive copying, invalidating iterators, and other surprises.

We propose that neither approach is the default and require the programmer to explicitly declare whether by-reference or by-copy is desired. There are two mechanisms for this. Either specifying "by-reference" for the entire lambda, or by specifying one of the two modes for each variable stored into the closure We discuss the latter mechanism first.

All local variables referred to in the body of the lambda function must be declared and initialized alongside the parameters of the lambda function. These declared local variables are what are stored in the closure. The following example defines a lambda function, where the closure stores a reference to the local variable sum, and stores a copy of a local variable factor:

```
double array[] = { 1.0, 2.1, 3.3, 4.4 };
double sum = 0; int factor = 2;
for_each(array, array + 4, <>(int n : double& s = sum, int f = factor) { s += f * n });
```

We refer to the part of the function signature that declares and initializes the member variables of the closure as the lambda function's *local variable clause*.

To clarify how local variables are handled, the above lambda function is behaviorally equivalent to the following function object:

```
class F {
  double& sum;
  mutable int factor;
public:
  F(int& sum, int factor) : sum(sum), factor(factor) {}
  auto operator()(int n) const −>decltype(make<double&>() += make<int&>() ∗ n) {
    return sum += factor ∗ n;
  }
};
F(sum, factor); // create the closure
```

The use of **decltype** operator is a bit complex, as the member variable names sum and factor are not in scope in the signature of **operator**(); instead of sum and factor, we use the expressions make<**double**&>() and make<**int**&>() that have, respectively, the same types as expressions sum and factor in the body of **operator**().

The syntax for declaring the variables to be stored into closures is the standard syntax for declaring and initializing variables. It follows that the **auto** mechanism [JSR06b] can be used, avoiding the need to specify the types of the local variables explicitly. For example, the above lambda function could alternatively be written as:

```
<>(n : auto& s=sum, auto f=factor) { s += f∗n }
```

We propose an additional shortcut: a single unqualified identifier, optionally preceded with &, that names a variable with a non-static storage duration can be used as an element the local variable clause. A single identifier, say x, is interpreted as the declaration **auto** x = x; preceding an identifier with & as in &x, is interpreted as **auto**& x = x. Here, x on the right refers to a variable in scope of the lambda definition, x on the left is the newly declared variable to be stored in the closure. With the shortcut, the above lambda can be written as:

```
<>(n : &sum, factor) { sum += factor∗n }
```

The variable **this** requires special handling. If a lambda function is defined in a member function, the body of the lambda function may contain references to **this**. To allow these references, **this** must be explicitly declared in the local variable clause. In the translation, references to **this** in the body of the lambda function are retained to refer to the object in whose member function the lambda was defined, not the just generated closure object. We can interpret **this** in the local variable clause as **auto** this_ = **this**, where this_ is some unique name. All references to **this** will be then translated to references to this_. The type of this_ will be appropriately qualified depending on the cv-qualifiers of the member function the lambda is defined in. Also, the newly generated closure should be a **friend** of the enclosing class to allow access to its private members. The following example demonstrates the handling of **this**, and the necessity of making the lambda function a **friend** of the enclosing class:

```
class A {
  vector<int> v;
  ...
public:
  void zero_all(const vector<int>& indices) {
    for_each(indices.begin(), indices.end(), <>(i : this) { this−>v[i] = 0 });
  }
};
```

Requiring explicit declaration of the variables that are to be stored into closure has the benefit, over the previous proposals, that the programmer is *forced* to express his or her intention on what storage mechanism should be used for each local variable. The disadvantage is verbosity. The full variable declaration syntax allows storing values computed from one or more local variables into closures. For example:

```
<>(n : auto &s = sum, auto f = factor∗2) { s += f∗n } // f is a computed value
```

We suspect a common use of lambda functions is as function objects to standard algorithms and other similar functions, where the lifetime of the lambda function does not extend beyond the lifetime of its definition context. In such cases, it is safe to store the environment into a closure by reference. For this purpose, we suggest syntax that allows the "by-reference" declaration to be made for the entire lambda at once, and not requiring explicitly listing variable names in a local variable clause. Rewriting our previous example some, the two calls to for_each below are equivalent:

```
double array[] = { 1.0, 2.1, 3.3, 4.4 };
double sum = 0; int factor = 2;
```

```
for_each(array, array + 4, <>(n : &sum, &factor) { sum += factor ∗ n });
for_each(array, array + 4, <&>(n) { sum += factor ∗ n });
```

Note that when the closure stores no copies of variables in the context of the lambda definitions, a different translation is possible. The resulting function object could only store a single pointer to the stack frame where the lambda is defined, and access individual variables via that pointer. We continue, however, to describe the semantics of lambda functions via a translation to function objects with one member variable for each distinct variable that is referred to in the body of the lambda function. Nevertheless, any implementation resulting in the same behavior should be allowed. Section 7.1 discusses the <&> form further, and argues for requiring the "single pointer to environment" closures described above in order to have a fixed size binary representation of lambda functions.

We summarize the rules regarding references to variables defined outside of the lambda function:

- The body of the lambda function can contain references to the parameters of the lambda function, to variables declared in the local variable clause, and to any variable with static storage duration.
- If the lambda function is declared with the <&> form, additionally references to all local variables in scope at the definition of the lambda function are allowed.

4 Type checking lambda functions

The bodies of non-generic lambda functions can be type checked immediately at the point of their definition. Generic lambda functions, however, cannot be type checked just by seeing the definition of the lambda function, as its parameter types are not known. The following two functions demonstrate:

```
<>(int i){ i + 1 }
<>(i) { i + 1 }
```

The body of the first function can be type checked, knowing that i is of type **int**. The second function provides no information on what is the type of i, and thus the expression i + 1 in its body cannot be type checked.

To be able to type check a generic lambda function, we must first know what will be its argument types. This information can be obtained if the lambda function is called:

```
<>(i) { i + 1 } (5)
```

We can now deduce that i will be of type **int**, and type-check i + 1 using that information. This is comparable to C++'s unconstrained templates: a template function is type checked only after it is instantiated. In the above example, the late type checking is not a modularity problem, because the use follows immediately after the definition. Lambda-functions, however, are seldom called where they are defined. More often than not, they will be bound to a variable or a function parameter, and invoked later in a different scope. We thus want to effect type checking when a lambda is bound to variable that can be used to pass the lambda to a different scope. From the type of this variable the type checker must extract the types of the arguments that the lambda function may be called with.

The type of a lambda function is some unique class type, known only to the compiler. This means that it is not possible to declare a variable, or a function parameter, with the exact type of a lambda function. Lambda functions can, however, be bound to function parameters whose types are template parameters, or to variables declared with **auto** in order to have their type deduced from their initializer expression. We focus on the former case; the latter is discussed in Section 4.1. Also, we only discuss constrained templates—unconstrained templates provide no useful information for type checking prior to instantiation time.

Consider the following code where a lambda function is bound to a parameter of a function template:

```
template <typename F, typename T>
requires R
void foo(F f, T t) { f(t); }

foo(<>(i) { i + 1 }, 5);
```

First, for the function foo to type check, some constraint in R must state that an object of type F must be callable with an argument of type T. An example of such a constraint is std::Callable1<F, T>, which brings with it the requirement result_type **operator**()(F, T). When type checking the above call to foo, after deducing F and T with template argument deduction, the constraints of R contain the information with which argument types the lambda function may be called in the body of foo. In the call above, T is deduced to **int**, leading to the requirement result_type **operator**()(F, **int**). This essentially fixes the argument types of the lambda function, turning it from a generic to a non-generic function. Section 5 describes in details how this kind of type checking can be attained.

4.1 Type checking and auto variables

Lambda functions can be used as initializer expressions of variables declared with **auto**. For example:

```
auto f = <>(i) { i + 1 };
```

This kind of binding a lambda function to a name reveals no new type information, and does not allow type-checking of the body. With **auto** variables, the definition of a lambda function can be separated from the site where enough information is obtained to be able to type check the function's body. For example:

```
auto f = <>(i) { i + 1 };
// many lines of code here
f(5);
```

This is unavoidable. However, the location where type-checking becomes possible must necessarily be in the lexical scope of the definition. In particular, without resorting to unconstrained templates, it is not possible to return a lambda function as the result of the function defining it or passing it out of the function via a reference parameter *without coercing* the lambda function into a type that allows the lambda function's body to be type-checked. For example, in the following function, the constructor of std::function<**int**(**int**)> provides the information that the lambda function's parameter type is **int**.

```
std::function<int(int)> counter() {
  return <>(x: int c = 0) −>int { c += x };
}
```

5 Full translation semantics

We define the semantics of lambda functions via a translation to function objects. Implementations should not, however, be required to literally carry out the translations. We first explain the translation in a case where the body of the lambda function is a single expression, the return type is not specified, the definition resides in a generic context, and the parameter and closure member types are explicitly specified. Variations are discussed later.

The left-hand side of Figure 1 shows a lambda function in the context of two nested templates. The right-hand side shows the translation. For concreteness of presentation, we fix the example to use two parameters and two local variables. The lambda function is on lines a8–a12. We have included a generic context, the two template parameter lists (with their requirements) on lines a1 and a5. The translation of a lambda function consists of the generic context created with nested template classes in namespace scope (see lines b3 and b6); the closure class unique (line b9), generated into the translated generic context; and a call to the constructor of unique to replace the lambda function definition (starting from line b34). The following list describes the specific points of the translation:

1. It may not be possible to generate the closure class in the point of its definition. In particular, the closure class must be usable as a template argument, and cannot thus be a local class. Therefore, the closure class is to be generated into the innermost non-function scope immediately preceding the site of the definition of the lambda function. All template context, possibly consisting of several nested templates, "lost" in moving up the definition must be replicated in the translation. The replication of the entire context of template parameters and their constraints is necessary to avoid the so called "concept map dropping" problem, described in Section 9. In our example, the template parameter list of the source templates (lines a1 and a5) along with their constraints are copied as such to the target templates (lines b3 and b6); there is no need to rename template parameters.

2. The formal template parameters of the context of the lambda function's definition site are used as template arguments to instantiate, respecting the nesting structure, the template context around the generated closure (see line 34). Again, no change in parameter names is necessary.

```
template <C1 A, typename B>
requires C2<B>
void outer(...) {
  ...
  template <D1 X, D2 Y>
  class inner {
    ...
    <>(ptype1 param1,
        ptype2 param2)
        : vtype1 var1 = init1,
          vtype2 var2 = init2)
    { body }
```

```
// generated to the innermost non-function scope from
// the definition site of the lambda function
template <C1 A, typename B> requires C2<B>
struct unique_outer {

  template <D1 X, D2 Y>
  struct unique_inner {

    class unique {
      vtype1-t var1;
      vtype2-t var2;
        ...
    public: // But hidden from user
      unique(vtype1-t-param var1, vtype2-t-param var2)
        : var1(var1), var2(var2) {}

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

      auto operator()(ptype1 param1, ptype2 param2) const
      ->decltype( body-t-ret )
      { return body-t; }
};

template <C1 A, typename B>
requires C2<B>
void outer(...) {
  ...
  template <D1 X, D2 Y>
  class inner {
      ...
  // generated to exactly where the lambda function is defined
  unique_outer<A, B>::template
  unique_inner<X, Y>::
  unique(init1, init2))
```

Figure 1: Example translation of lambda functions. The ellipses are not part of the syntax but stand for omitted code.

3. The function object (here named unique) implementing the lambda function is generated inside the templates that create the template context.

   The data of the closure is stored in the member variables of unique. In the example, two variables are stored in the closure: var1 and var2. The type vtype1−t is the type obtained for var1 in interpreting vtype1 var1 = init1 as a variable declaration. Unless **auto** is used, vtype1−t is equal to vtype1; vtype2−t is obtained analogously. Note that vtype1 and vtype2 may not be usable directly in the signature of the function call operator unique, because they may contain locally defined typedefs, or names of local classes. Expressing types in a way that avoids references to local classes or typedefs should pose no major problem to compilers. Non-reference non-const members are to be declared mutable—this is to allow updating member variables stored in the closure even though the function call operator of the closure is declared **const**. For an example, see the counter function in Section 4.1.

4. The closure's constructor has one parameter for each member variable. The parameter types vtypeN−t−param are obtained from vtypeN−t types by adding **const** and reference, leaving however non-const reference types intact. This constructor need not be exposed to the user.

   The closure also has copy and move constructors with their canonical implementations.

   Closure classes do not have a default constructor or an assignment operator.

5. The parameter types of the function call operator are those defined in the lambda function's parameter list. The return type is obtained as **decltype**(body−t−ret) where body−t−ret is obtained from the lambda function's body with a minor translation. The original body may contain references to the member variables var1 and var2, which are not in scope in signatures of member functions. Each reference to varN must thus be replaced with a valid expression with the type that varN has in the function body, such as make<vtypeN−t&>(). Reference is added because member access operators via, possibly implicit, **this** return an lvalue.

6. A lambda function's body may only contain references to its parameters, variables defined in the local variable clause, and any variable with static storage duration. A minor translation for the body is necessary if the lambda function is defined in a member function and contains references to **this**. The translation was described in Section 3.2.

7. The names of all the classes generated in the translation should be unique, different from all other identifiers in the entire program, and not exposed to the user.

The above example translation was a case where the body of the lambda function is a single expression, the return type is not specified, the definition resides in a generic context, and the parameter types are explicitly specified. If the return type is specified explicitly, that return type is used as the return type of the function call operator of the closure. Note that textually the type expression may not be directly usable as the return type, as it may contain references to local classes or typedefs, but a translation avoiding these pitfalls should pose no problems. The case where the body of the lambda function is a compound statement, instead of a single expression, is only trivially different to the translation described above. Implicit parameter types, however, require more complex handling, discussed in Section 6.

6 Generic lambda functions

This section describes how polymorphic, or generic, lambda functions can be type-checked at the scope of their definition. The following outlines the process:

1. When the compiler encounters a lambda definition, a closure class is generated, according to the translation described in Section 5; the function call operator is, however, not yet generated. Consequently, an isolated definition of a polymorphic lambda function does not trigger type checking of the lambda function's body. For example, the definition of the lambda function below does not cause a type error:

   ```
   int* ptr;
   <>(i){ i + ptr + ptr };
   ```

   Similarly, binding a lambda function to an **auto** variable does not trigger type-checking:

```
auto still_no_error = <>(i){ i + ptr + ptr };
```

2. There are three different operations that a lambda function can be subjected to after its definition: it can be called, its type can be bound to a constrained template parameter, or its type can be bound to an unconstrained template parameter. We discuss each of those in turn:

   (a) **Calling a lambda function.** When compiling an invocation to a lambda function, argument types to it become known. Consider the following call:

   ```
   int prod = 1;
   auto cumul_product = <>(x : auto p& = prod) { p *= x }
     ...
   cumul_product(5);
   ```

   The type of the argument 5 is **int**, which we use as the type of the lambda function's parameter x. We must also select the parameter passing mode. If the type of an argument is T, the parameter type is T&&. For example, above, the parameter type is **int**&&. In the call below, the parameter type is **const int**&&:

   ```
   const int fact = 10;
   cumul_product(val);
   ```

   Hence, when encountering a call to a lambda function, the type checker's tasks are to

   i. deduce the argument types;
   ii. generate a function call operator with the deduced argument types and with the body of the lambda function, and inject that to the closure class; and
   iii. to type check the body of the function call operator.

   For example, the code below shows the generated function call operators for the cases of the argument type being deduced to **int**&& and **const int**&&:

   ```
   auto operator()(int&& x) const −>decltype(make<int&>() *= x) { return p *= x; }
   auto operator()(const int&& x) const −>decltype(make<int&>() *= x) { return p *= x; }
   ```

   As the above example demonstrates, a single lambda function can be invoked at more than one call sites, with different argument types. Thus a closure class can end up containing more than one function call operator, all with equivalent bodies.

   (b) **Binding a lambda function to a constrained template argument.** In the common case, a lambda function is passed to another scope prior to invoking the lambda function. Consider the following example:

   ```
   std::vector<T> v; // T is some type parameter
   T factor = ...;
   std::transfrom(v.begin(), v.end(), v.begin(), <>(x : auto p = factor) { x *= p });
   ```

   In this case, the type checker proceeds to do overload resolution in the normal manner. A possible declaration of the transform function in the standard library is as follows:

   ```
   template<InputIterator I1, typename O1, typename F>
   requires OutputIterator<O1, Callable1<F, I1::value_type>::result_type>
   O1 transform(I1 first, I1 last, O1 result, F f) {
     for ( ; first != last; ++first, ++result)
       *result = f(*first);
     return result;
   }
   ```

   After template argument deduction, the constraints need to be checked. This takes place as defined in the concepts proposal [GS06], except for function call operator requirements, which need to be handled specially. Denote the type of the lambda function in the call as L. One of the constraints is thus

```
Callable1<L, InputIterator<vector<T>::iterator>::value_type>
```

where InputIterator<vector<T>::iterator>::value_type can be resolved to T. The definition of Callable1 is as follows:

```
auto concept Callable1<typename F, typename T1> {
  typename result_type;
  result_type operator()(F&, T1);
}
```

The requirement of a function call operator then becomes: result_type **operator**()(L&, T). This information can be used to generate the new function call operator into the closure class (make<T&>() comes from the type of the member variable p):

```
auto operator()(T x) const −>decltype(x ∗= make<T&>()) { return x ∗= p; }
```

After injecting this function call operator, the operator's body can be type checked. Also, due to now knowing the argument types, the return type of the function call operator is known as well. The callability requirement may place constraints on the return type of the function object type, so the closure should be now also type checked against the function call operator constraint; in the above example this means checking that L with the newly generated function call operator and T are a model of Callable1<L, T>.

Note that there can be more than one function call operator requirement on a single template parameter. One way to deal with this is to inject a function call operator into the closure class for each call with different argument types.

(c) **Binding a lambda function to an unconstrained template argument.** No checking takes place, therefore the function call operator is not generated either. Generation and checking only takes place when the template is instantiated with concrete non-generic types. For example:

```
template <typename F, typename T>
void forwarder(const F& f, const T& t) { return f(t); }

void forward_to_forwarder(int k) {
  forwarder(<>(i){ i + ptr + ptr }, k);
}
```

The error message is obtained only when instantiating the body of the forwarder function, when the types of the arguments to the lambda function become known.

7 Binary interface

A closure is of some compiler generated class type. To pass lambda functions to non-template functions, one can use the std::function, or other similar library facilities. For example, the doit function below accepts any function or function object F, including a closure, that satisfies the requirement Callable<F, **int**, **int**>:

```
void doit(std::function<int (int, int)>);
```

The doit function is non-generic, and could be placed, e.g., in a dynamically linked library. The following shows a call to doit, passing a lambda function to it:

```
int i;
 ...
doit(<>(x, y : i) { x + i∗y });
```

The std::function template has a fixed size, and it can be constructed from any MoveConstructible function object which satisfies the callability requirement of the std::function instance. A typical implementation of function uses so called "small buffer optimization" to store small function objects in a buffer in the function itself, and allocate space for larger ones dynamically, storing a pointer in the buffer.

7.1 The <&> form

Access to variables declared in the lexical scope of a lambda function can be implemented by storing a single pointer to the stack frame of the function enclosing the lambda. This lends to representing closures using two pointers: one to the code of the lambda function, one to the enclosing stack frame (known as the "static link"). As discussed above, this representation is only safe for lambda functions that do not outlive the functions they are defined in.

If the "two-pointer" representation is mandatory, lambda functions can be given a light-weight binary interface, even more so than using std::function: lambda functions can be passed in registers, and they can be copied bitwise. We put forward the suggestion that lambda functions defined using the <&> syntax are required to use this representation, and that their types are instances of a "magic" template "std::nested_function". To extend the C++ type system with a full-fledged new "lambda-function" type would be rather drastic, which is why we suggest std::nested_function.

The definition of std::nested_function would be similar to that of std::function:

```
template<class MoveConstructible R, class MoveConstructible... ArgTypes>
class function<R(ArgTypes...)>
struct nested_function {
  R operator()(ArgTypes...) const;
  // copy and move constructors
};
```

A function expecting an std::function parameter accepts any function pointer or function object type, assuming the function or function object is callable with the required signature; a function expecting a std::nested_function only matches lambda functions defined using the <&> form. A binary library interface can provide overloads for both types. For example:

```
void doit(std::function<int(int, int)> f);
void doit(std::nested_function<int(int, int)> f);
```

This overload sets allows calls with a function, function object, or lambda function, and takes advantage of the latter overload when called with a <&> form lambda function; in a call to doit where the parameter is a lambda defined with the <&> form, the match to the first requires a user-defined conversion via the templated constructor of std::function, whereas the latter is a direct match.

The code generation for the closures implemented as instances of nested_function must occur in the template context of the definition of the lambda function (cf. the replicating of the template context in the translation) to avoid the concept map dropping problem discussed in Section 9.

8 About particular design choices

This section discusses some design choices that may not get sufficient attention in the main body of the proposal.

8.1 Statements vs. expressions as the lambda body

In what is being proposed, the body of the lambda is either of the form: { *primary-expression*} or *compound-statement*. This is a bit subtle—a semicolon makes the difference. For example, the following lambda expression would be OK:

```
<>(x, y) { x + y }
```

whereas the one below an error:

```
<>(x, y) { x + y; }
```

The correct writing of the latter, using a compound statement, is:

```
<>(x, y) −>decltype(x + y) { return x + y; }
```

As rationale for allowing both forms, we feel that restricting lambda function bodies to only compound statements is unnecessary verbose in simple cases. On the other hand, disallowing statements in the function bodies is too restrictive; consider the following example:

```
for_each(a.begin(), a.end(),
  <>(x) ->void { if (cond(x)) then cout << x; else dosomethingelse; })
```

8.2 Direct initialization in local variable clause

Direct initialization syntax is not allowed in the local variable clause. For example, the following is an error.

```
<>(x : vector<int> v(it1, it2) ) { ... }
```

Direct initialization would complicate the translation semantics some. In the above example, it1 and it2 each would need a parameter slot in the constructor of the closure object. Their types need to be deduced either by performing overload resolution to the constructor of vector<**int**> and copying the parameter types from the best matching overload, or by leaving them as template parameters (T&&). Neither of this is necessary if direct initialization syntax is not supported.

8.3 Alternative syntaxes

Other syntaxes are possible. Instead of "<>" and "<&>", "/" and "/&" have been suggested. Also, whether the parentheses around the lambda parameter list and local variable clause are necessary is worth considering. The following example demonstrates:

```
/x, y { x + y }
/x, y : &a, b { x + y + a + b }
/& x, y { x + y + a + b }
```

Another suggestion has been to use ";" instead of ":" as the separator between parameter list and local variable clause—this would be similar to the use of semicolon in **for** statements. For example:

```
<>(x, y ; &a, b) { x + y + a + b }
```

8.4 No "per-variable" switch for by-copy and by-reference

The <&> is a default for "store all references to local variables by reference". We do not propose a similar default for "store all reference to local variables by copy." When declaring closures that stores copies of data, we find it desirable that the programmer is forced to be explicit of what is being copied into the closure. Assume that <c> declares a lambda where the local environment is copied into the closure by default. Compare then the two lambda functions below:

```
vector<int> v; // assume this is a very large vector
int index;
...
<c>(x) { v[index] + x }
<>(x : auto vi = v[index] ) { vi + x }
```

The former draws in the entire vector into the closure, whether or not this was what the programmer desires. The latter lambda function only stores the one value that is necessary.

9 Concept map dropping

The concept system provides an "escape hatch" from constrained templates to unconstrained templates. With the current specification of concepts [GS06] the escape hatch exhibits somewhat brittle and surprising behavior. Using the escape hatch obviously nullifies all guarantees about separate type checking, leading to instantiation time errors if a template's implicit requirements, induced by what operations are used in the body of the template, are not met. However, even if all the implicit constraints are met in the instantiation context of an unconstrained template, a compiler error can still occur. Even worse, the code can compile and manifest surprising behavior at run-time. This can occur if so called *syntax remapping* is used in concept maps. The minimal example below explains the issue: *concept maps being dropped*.

```
concept C<typename T> { void op(T); };

concept_map C<int> { void op(int) {} };

template <typename T> void f(const T& t) { op(t); } // unconstrained template
template <C U> void g(const U& u) {
  op(u); // OK
  f(u); // error
}
...
g(1);
```

Here, a constrained function template g calls an unconstrained function template f. This is allowed, and all checking is to be delayed until instantiation time. As the call to f is the sole contents of g's body, g type-checks. Furthermore, one should expect no instantiation time errors as f seems to only use functionality that g's constraints guarantee. The call g(1), however, produces the error:

```
test-simple.cpp: In function 'void f(const T&) [with T = int]':
test-simple.cpp:8:   instantiated from 'void g(const U&) [with U = int]'
test-simple.cpp:11:   instantiated from here
test-simple.cpp:7: error: 'op' was not declared in this scope
```

The reason is that when instantiating unconstrained templates, concept map declarations have no effect. Thus, even though op(u) works, resolving to C<**int**>::op(**int**), the same invocation fails when placed from the unconstrained template f<**int**>—the concept map for C<**int**> was dropped. Thus, generating lambda functions as unconstrained templates would result in lambdas exhibiting the problems of concept map dropping.

10 Acknowledgements

We are grateful for help and comments by Dave Abrahams, Matt Austern, Peter Dimov, Gabriel Dos Reis, Doug Gregor, Andrew Lumsdaine, Valentin Samko, Jeremy Siek, Bjarne Stroustrup, Herb Sutter, Jeremiah Willcock, Jon Wray, and Jeffrey Yasskin.

References

[GS06] Douglas Gregor and Bjarne Stroustrup. Concepts (revision 1). Technical Report N2081=06-0151, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2006.

[Hin06] Howard Hinnant. A proposal to add an rvalue reference to the C++ language: Proposed wording revision 2. Technical Report N1952=06-0022, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, January 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1952.html`.

[JSR06a] Jaakko Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype (revision 6): proposed wording. Technical Report N2115=06-0185, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, November 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2115.pdf`.

[JSR06b] Jaakko Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Deducing the type of variable from its initializer expression (revision 4). Technical Report N1984=06-0054, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, April 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1984.pdf`.

[Sam06] Valentin Samko. A proposal to add lambda functions to the C++ standard. Technical Report N1958=06-0028, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2006. `www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n1958.pdf`.

[WJG$^+$06] Jeremiah Willcock, Jaakko Järvi, Douglas Gregor, Bjarne Stroustrup, and Andrew Lumsdaine. Lambda functions and closures for C++. Technical Report N1968=06-0038, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf`.