Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 3)

Document no: N2529=08-0039

Jaakko Järvi* Texas A&M University
John Freeman Texas A&M University
Lawrence Crowl Google Inc.

2008-02-04

1 Introduction

This document describes *lambda expressions*, reflecting the specification that was agreed upon within the evolution working group of the C++ standards committee in the 2007 Kona meeting. The document is a revision of N2487 [JFC07b], N2413 [JFC07a], and N2329 [JFC07c]. N2329 was a major revision of the document N1968 [WJG+06], and draw also from the document N1958 [Sam06] by Samko. The differences to N2487 are:

- Added wording that specifies the nested_function template—type of closures that store all local variables by reference.
- Corrected erroneous specification of closure's move constructor, suggested by Howard Hinnant.
- Small disambiguating change to grammar suggested by Clark Nelson [Nel08].

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 specification (not necessarily the implementation) of the proposed features relies on several future additions to C++, some of which are already in the working draft of the standard, others likely candidates. These include the **decltype** [JSR06b] operator, new function declaration syntax [JSR06a, Section 3][Mer07], and changes to linkage of local classes [Wil07].

The proposed wording in this document is partial in that the library sections of the specification of the nested_function class template are not yet written.

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 what, e.g., in the context of functional programming languages is known as a *closure*. 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.

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 thus 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 = ....
....
double u_limit = 1.1 * min_salary;
std::find_if(employees.begin(), employees.end(),
             <&>(const employee& e) (e.salary() >= min_salary && e.salary() < u_limit));
```

3 About non-generic and generic lambda functions

The lambda expression:

```
<&>(const employee& e) (e.salary() >= min_salary && e.salary() < u_limit)
```

is *monomorphic* because the types of its parameters are explicitly specified. Here, the type of the only parameter e has type **const** employee&. A *polymorphic* of *generic* version of the same expression would be written as:

```
<&>(e) (e.salary() >= min_salary && e.salary() < u_limit)
```

The latter form requires that the parameter types are deduced (from the use of the lambda expression), and have a substantially higher implementation cost, as discussed in [JFC07c]. We do not propose generic lambda functions for C++0x.

4 Proposal

In the following, we introduce the proposed features informally using examples of increasing complexity.

4.1 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:

```
<>(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**. In this example, the body of the lambda function is a single expression, enclosed in parentheses. 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).

The explicit specification of the return type is also allowed; 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; }
```

There are several reasons for requiring the return type to be specified explicitly in the case where the body consist of a block statement. 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, or rules similar to those of the conditional operator could be applied, which is potentially complicated. Second, implementing return type deduction from a statement block may be non-trivial. The return type is no longer dependent on a single expression, but rather requires analyzing a series of statements, possibly including variable declarations etc. Third, if a lambda expression is passed as an argument to an overloaded function, and its return type expression contains template parameters, the return type expression may have to be instantiated to determine whether a particular function matches. To avoid hard errors during overload resolution, certain errors in the return type expression should fall under the SFINAE rules. Whether an arbitrary block type checks or not as a SFINAE condition is not feasible, nor desirable.

The semantics of lambda functions are defined by a translation to function objects. For example, the last lambda function above behaves as the function object below. The proposed translation, described in Section 5, is somewhat more involved.

```
class F {
public:
  F() {}
  auto operator()(int x, int y) const -> int {
    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 be a function parameter declaration with a non-abstract declarator; a type without a parameter name is disallowed to preserve upwards compatibility to polymorphic lambda functions, where a single identifier is interpreted as the name of the parameter, not a type.
- The body of the lambda function can either be a single expression enclosed in parenthesis 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.

4.2 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. Such variables are commonly called *free variables*. An earlier proposal [WJG+06] called for storing free 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.

In N2329 [JFC07c] we proposed that neither approach is the default and require the programmer to explicitly declare for each free variable whether by-reference or by-copy is desired. We keep this as the most basic form of lambda expressions—two other forms, a "by-reference" and "by-copy" as the default for the entire lambda expression, are additionally provided for more succinct definition of certain lambda expressions. We discuss the basic form first.

All local variables referred to in the body of the lambda function (but defined outside of it) must be declared 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, <>(double d) : [&sum, factor] ( sum += factor * d ));
```

We refer to the part of the function signature that declares the member variables of the closure as the lambda expression's *local variable clause*. The local variable clause is a comma separated list of unqualified identifiers, optionally preceded with &, that name a variable with a non-static storage duration. The closure stores references to objects whose identifiers are declared with & and copies of objects whose identifiers are declared without &. The optional return type of the lambda expression precedes the local variable clause.

To clarify how local variables are handled, the above lambda function behaves as the following function object:

```
struct F {
  double& sum;
  mutable int factor;

  F(int& sum, int factor) : sum(sum), factor(factor) {}

  auto operator()(double d) const -> decltype(fake<double&>() += fake<int&>() * d) {
    return sum += factor * d;
  }
};
F(sum, factor); // create the closure
```

The return type expression cannot simply be **decltype**(sum += factor * d) because the member variables sum and factor cannot be used outside the body of the **operator**()[1]. We thus replace each use of sum and factor, respectively, with expressions that have the same types as these variables — the expression fake<T>() has the type T for any type T. The fake function can be defined as:

```
template <typename T> T fake();
```

4.2.1 Handling this pointer

If a lambda function is defined in a member function, the body of the lambda function may contain occurrences of **this** (either implicit or explicit). After the translation, these occurrences should refer to the

[1] An *id-expression* that denotes a non-static member variable or function of a class can appear outside of a class body as an operand to **decltype**, but not as a proper subexpression of an operand to **decltype** (see [Bec07, §5.1(11)]).

object in whose member function the lambda was defined, not to the just generated closure object. First, to allow references to **this**, one includes **this** in the local variable clause. The effect is that some unique member variable, call it __this, will be added to the closure object to store the value of **this**. All references to **this**, including implicit member access operator calls that do not mention **this** explicitly, are 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 have access to the private members of its enclosing class. The following example demonstrates the use of **this** in the local variable clause:

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

After translating the lambda expression to a function object the above code becomes:

```
class A {
  vector<int> v;
  ....
public:
  void change_sign_all(const vector<int>& indices) {
    class G {
      A* __this;
    public:
      G(A* __this) : __this(__this) {}
      auto operator()(int i) const -> decltype(fake<A*&>()-> v[i] = fake<A*&>()-> v[i]) {
        return __this-> v[i] = v[i];
      }
    };
    for_each(indices.begin(), indices.end(), G(this));
  }
};
```

4.2.2 Alternative forms of lambda expressions

Requiring the programmer to explicitly declare the variables to be stored in the closure has the benefit that the programmer is *forced* to express his or her intention on what storage mechanism to use for each local variable. The disadvantage is verbosity. 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. Thus, we suggest syntax that allows the "by-reference" declaration to be made for the entire lambda at once, instead of 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, <>(double d) : [&sum, &factor] ( sum += factor * d ));
for_each(array, array + 4, <&>(double d) ( sum += factor * d ));
```

The <&> form establishes "by-reference" as the default mode for storing local variables in the closure, which can be overridden in the local variable clause for individual variables. A third form is also provided, with

<=> starting the lambda definition. This form is opposite to the <&> form in that "by-copy" is the default mode for storing local variables, and can be overridden in the local variable clause. For example, the following three lambda expressions are all equivalent:

```
<>(double d) : [sum, &factor] ( sum += factor * d )
<&>(double d) : [sum] ( sum += factor * d )
<=> (double d) : [&factor] ( sum += factor * d )
```

Note that when the closure stores no free variables as copies, a different translation is possible: the closure function object can store a single pointer to the stack frame where the lambda function is defined, and arrange access to individual free variables via that pointer. Section 6.1 discusses this implementation in more detail. We continue, however, to describe the semantics of lambda functions via a translation to function objects with one member variable for each distinct free variable.

We summarize the rules regarding free variables:

- The body of the lambda function can refer 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 defined in a member function of some class, call it A, and the local variable clause contains the keyword **this**, the body of the lambda function can additionally refer to members of A and contain occurrences of **this**.
- If the lambda function is declared with the <&> or <=> form, its body can additionally refer to local variables with automatic storage duration that are in scope where the lambda function is defined.

5 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 and the return type is not specified. Variations are discussed later.

The left-hand side of Figure 1 shows a lambda function, the right-hand side its translation. For concreteness of presentation, we fix the example to use two parameters and two free variables. The translation of a lambda function consists of the closure class unique (line b3), generated immediately before the statement or declaration where the lambda expression appears, and a call to the constructor of unique to replace the lambda function definition (on line b21). The following list describes the specific points of the translation:

1. The closure object stores the necessary data of the enclosing scope in its member variables. In the example, two variables are stored in the closure: var1 and var2. The types vtype1−t and vtype2−t are the types of var1 and var2, respectively, in the enclosing scope of the lambda expression, augmented respectively with amp1 and amp2 which are either & or empty. Non-reference non-const members are to be declared **mutable**—this is to allow modifications to member variables stored in the closure even though the function call operator of the closure is declared **const**.
2. The closure's constructor has one parameter for each member variable, whose types vtypeN−t−par are obtained from vtypeN−t types by adding **const** and reference to all non-reference types. This constructor should not be exposed to the user.
3. The closure has copy and move constructors with their canonical implementations.
4. Closure classes should not have a default constructor or an assignment operator.
5. If the lambda function is defined in a member function and its body contains references to **this** or member access operations that do not explicitly mention **this**, the translation described in Section 4.2.1 is applied to obtain body−t from body.

```
a1  <>(ptype1 par1, ptype2 par2)
a2        : [amp1 var1, amp2 var2]
a3        ( body )
```

```
b1  // generated immediately before the statement
b2  // that the lambda expression appears in
b3  class unique {
b4      vtype1-t var1;
b5      vtype2-t var2;
b6      ....
b7  public: // But hidden from user
b8    unique(vtype1-t-par var1, vtype2-t-par var2)
b9      : var1(var1), var2(var2) {}
b10 public: // Accessible to user
b11   unique(const unique& o) : var1(o.var1), var2(o.var2) {}
b12   unique(unique&& o) : var1(static_cast<vtype1-t&&>(o.var1)),
b13                              var2(static_cast<vtype2-t&&>(o.var2)) {}
b14
b15   auto operator()(ptype1 par1, ptype2 par2) const
b16     ->decltype( body-t-ret )
b17   { return body-t; }
b18 };
b19
b20 // generated to exactly where the lambda function is defined
b21 unique(init1, init2)
```

Figure 1: Example translation of lambda functions.

6. The parameter types of the function call operator are those defined in the lambda function's parameter list. The return type is defined as **decltype**(body−t−ret) where body−t−ret is obtained from body−t by translating all occurrences of free variables to some expressions that have the same type and the same l-/r-valueness as the variable the occurrence refers to. With the help of, for example, the fake function (see Section 4.2), a variable of type T is translated to fake<T&>(). Occurrences of __this are subject to this translation as well.

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 and the return type is not 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, except that member access operations and references to variables is in the local variable clause are translated as described above in items 5 and 6. 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.

6 Binary interface

A closure object is of some compiler generated class type the programmer cannot directly write. The type can only be obtained via template argument deduction. To bind a lambda function to a parameter of a non-template function or a variable, the variable's or parameter's type must thus have some other type to which the lambda expression's type is convertible. For example, instances of the std::function can serve as such types.

All instances of the std::function template have a fixed size and can be constructed from any MoveConstructible function object which satisfies the callability requirement of the particular instance of std::function. 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.

The doit function below demonstrates how std::function can be used with lambda functions. The argument to doit can be any function or function object F—such as a closure object—that satisfies the requirement Callable<F, **int**, **int**> and whose return type is convertible to **int**: The doit function is non-generic, and could thus be placed, e.g., in a dynamically linked library.

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

The following shows a call that passes a lambda function to doit:

```
int i;
 ....
doit(<>(int x, int y) : [i] ( x + i*y ));
```

6.1 Closures with no by-copy variables

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 mentioned in Section 4.2.2, this representation is 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 and efficient binary interface, even more so than with std::function; lambda functions can be passed in registers, and they can be copied bitwise. Closures that do not store any free variables by copy are required to use this representation. 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 can be similar to that of std::function:

```
template<MoveConstructible R, MoveConstructible... ArgTypes>
class nested_function<R(ArgTypes...)> {
   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 that store no free variables by copy. 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 lambda function that stores no free variables by copy; in a call to doit where the parameter is a lambda function defined with the <&> form, the match to the first overload requires a user-defined conversion via the templated constructor of std::function, whereas the latter is a direct match.

Note further, that lambda expressions resulting in closures with an empty environment could in principle be given function pointer types. For example, the type of the lambda expression <>(**int** x, **int** y) (x + y) could be defined to be **int**(*)(**int**, **int**). We have not considered all the implications of this design choice—if such an exploration were to be carried out and the feature found desirable, wording could be written that was not inconsistent with the current proposal.

7 Discussion on design choices and alternatives

Clark Nelson has pointed out that the <=> as the "lambda-introducer" is separated into tokes as <= >, whereas other forms of the lambda-introducer would start with the < token. By selecting a different syntax

for <=>, and possibly for the `<&>` form as well, such complication could be avoided. At least the following possibilities would be free of this complication:

<!> <∗> <+> <.> <^> <|> <∼>

Clark also points out that the fact that unnamed parameters are not allowed in the parameter list of a lambda function is inconsistent with other functions. The rationale for this restriction is upwards compatibility to polymorphic lambdas where the parameter types could be omitted. If it was allowed to leave out either one, the type or the parameter, ambiguities would arise. Clark notes, however, that with either of the restrictions below, the ambiguities can be prevented:

- a prohibition against using an in-scope typedef name as the name of a parameter of a polymorphic lambda.
- different syntax (different lambda-introducer) to introduce polymorphic and monomorphic lambda functions

8 Acknowledgements

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

References

[Bec07] Pete Becker. Working draft, standard for programming language C++. Technical Report N2369=07-0229, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2007.

[JFC07a] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda expressions and closures: Wording for monomorphic lambdas. Technical Report N2413=07-0273, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, September 2007.

[JFC07b] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda expressions and closures: Wording for monomorphic lambdas (revision 2). Technical Report N2487=07-0357, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, December 2007.

[JFC07c] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda functions and closures for C++ (Revision 1). Technical Report N2329=07-0189, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, June 2007.

[JSR06a] Jaakko Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype (revision 5). Technical Report N1978=06-0048, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, April 2006.

[JSR06b] 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`.

[Mer07] Jason Merrill. New function declarator syntax wording. Technical Report N2445=07-0315, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2007. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2445.html`.

[Nel08] Clark Nelson. Lambda look-ahead issue. C++ standard's committees reflector, message `c++std-core-12636, 2008-1-30`, 2008.

[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`.

[Wil07] Anthony Williams. Names, linkage, and templates (rev 1). Technical Report N2187=07-0047, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, March 2007. `www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2187.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`.

A Proposed wording

The proposed wording follows starting from the next page. Within the proposed wording, text that has been added will be presented in blue and underlined when possible. Text that has been removed will be presented in red, with strike-through when possible. The wording in this document is based on the C++0X draft, and uses its LaTeX sources. There are some dangling references in the final document, which will be resolved when merged back to the full sources of the working paper.

Text typeset as follows is not intended as part of the wording:

[Editorial note: Example of a meta comment.]

Chapter 5 Expressions [expr]

5.1 Primary expressions [expr.prim]

Primary expressions are literals, names, ~~and~~ names qualified by the scope resolution operator `::`, and lambda expressions.

primary-expression:
- *literal*
- `this`
- `(` *expression* `)`
- *id-expression*
- *lambda-expression*

id-expression:
- *unqualified-id*
- *qualified-id*

unqualified-id:
- *identifier*
- *operator-function-id*
- *conversion-function-id*
- `~` *class-name*
- *template-id*

5.1.1 Lambda Expressions [expr.prim.lambda]

lambda-expression:
- *lambda-introducer* `(` *lambda-parameter-declaration-list*$_{opt}$ `)` *exception-specification*$_{opt}$ *lambda-return-type-clause*$_{opt}$ *lambda-local-var-clause*$_{opt}$ `(` *expression* `)`
- *lambda-introducer* `(` *lambda-parameter-declaration-list*$_{opt}$ `)` *exception-specification*$_{opt}$ *lambda-return-type-clause* *lambda-local-var-clause*$_{opt}$ *compound-statement*

lambda-introducer:
- `< >`
- `< & >`
- `<= >`
- `< = >`

lambda-parameter-declaration-list:
- *lambda-parameter*
- *lambda-parameter* `,` *lambda-parameter-declaration-list*

lambda-parameter:
- *decl-specifier-seq declarator*

lambda-return-type-clause:
- `->` *new-type-id*
- `->` `(` *type-id* `)`

lambda-local-var-clause:
- `:` `[` *lambda-local-var-list* `]`

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

lambda-local-var:
`&`$_{opt}$ *identifier*
`this`

1 The execution of a *lambda-expression* results in a *closure object*. Calling the closure object executes the expression or statements specified within the lambda-expression body. The type of a closure object depends upon the form of the lambda-expression. Some such types are defined within this standard; others are implementation-defined.

Closure objects behave as function objects ([function.objects], 20.5), whose function call operator, constructors, and member variables are defined by the lambda expression's signature, body, and the context of the lambda expression.

2 In a lambda expression

`<>(` *lambda-parameter-declaration-list*$_{opt}$ `)` *exception-specification*$_{opt}$ *lambda-return-type-clause* *lambda-local-var-clause*$_{opt}$ *compound-statement*

denote the *lambda-parameter-declaration-list*$_{opt}$ as `P`, *exception-specification*$_{opt}$ as `E`, *type-id* or *new-type-id* in *lambda-return-type-clause* `R`, *lambda-local-var-clause*$_{opt}$ as `L`, and *compound-statement* as `B`.

— The potential scope of a function parameter name in *lambda-parameter-declaration-list* begins at its point of declaration and ends at the end of the lambda expression.

— A name in the *lambda-local-var-clause* shall be in scope in the context of the lambda expression, and shall be `this` or refer to a local object with automatic storage duration. The same name shall not appear more than once in a *lambda-local-var-clause*.

— The lambda expression shall not refer to `this` or a variable of automatic storage duration in the enclosing scope of the lambda expression, if it is not named within the lambda expression's *lambda-local-var-list*.

3 To define the meaning of the lambda expression, define an empty class with a unique name, call it `F`, immediately preceding the statement where the lambda expression occurs.

Let $\texttt{n}_1$ `, ...,` $\texttt{n}_k$ denote the names defined in the *lambda-local-var-clause*, where `this` has been translated to some unique name, call it `__this`. Let $\texttt{t}_i, i = 1,\ldots,k$ denote the (non-reference) types of objects named $\texttt{n}_i$ looked up in the context of the lambda expression; type of `__this` is the type of `this` pointer. If *lambda-local-var-clause* is empty, $k = 0$, and we take the sequences $\texttt{n}_i$ and $\texttt{t}_i$ to be empty as well. Define in `F` a private member variable named $\texttt{n}_i$ for each $i = 1,\ldots,k$. If the name $\texttt{n}_i$ was declared with `&` in `L`, the type of the member variable named $\texttt{n}_i$ is $\texttt{t}_i$`&`, otherwise the type is $\texttt{t}_i$ and the member variable is declared `mutable`.

4 Let `r` be the type denoted by the *type-id* `R`. Let $\texttt{u}_1$ `, ...,` $\texttt{u}_n$ be the types denoted by the list of *type-id*s in `E`, if `E` is not empty.

Define a public function call operator in `F`:

```
auto operator()(P) const E' -> R' B'
```

where `R'` is some *type-id* that denotes the type `r`; where `E'` is empty if `E` is empty and otherwise `E'` is an *exception-specification* whose *type-id-list* consist of *type-id*s that denote the types $\texttt{n}_1$ `, ...,` $\texttt{n}_k$; and where `B'` is obtained from `B` by applying the following transformations:

— If the lambda expression is within a non-static member function of class `X`, perform name lookup. Transform all accesses to non-static non-type class members of the class `X` or of a base class of `X` that do not use the class member access syntax (5.2.5) to class member access expressions that use `(*this)`, as specified in ([class.mfct.non-static], 9.3.1).

— Transform all occurrences of `this` to `__this`.

5 `F` has an implicitly-declared copy constructor. Define a public move constructor for `F` that performs a member-wise move. The assignment operator in `F` has a *deleted definition*. Define no other public members to `F`. [EDITORIAL NOTE: The wording should include a reference to the definition of "move constructor". That definition does not seem to exist, though "move constructor" is mentioned in several places in the draft.]

6 Replace the lambda expression with an object of type `F` where each member named $\texttt{n}_i$ has been initialized with the variable named $\texttt{n}_i$ and the member `__this` initialized with `this`.

Denote the constructed `F` object `fo`. The meaning of the lambda expression is that of `fo`, except for the following:

— The size and alignment characteristics of the closure object are unspecified. In particular, they are not guaranteed to be those of objects of class `F`.

— If one or more variables in the local variable clause are declared with `&`, the effect of invoking a closure object, or its copy, after the innermost block scope of the context of the lambda expression has been exited is undefined.

— A closure object defined by a lambda expression with a local variable list consisting only of references shall have type `std::nested_function<R(P')>` (20.5.17), where `P'` is the comma separated list of types of the parameters in `P`. [*Note:* This requirement effectively means that such closures must be implemented via a "function pointer and static scope pointer" pair rather than by construction of a unique class type. — *end note*]

7 The meaning of a lambda expression

> `<&>(` *lambda-parameter-declaration-list*$_{opt}$ `)` *exception-specification*$_{opt}$ *lambda-return-type-clause* *lambda-local-var-clause*$_{opt}$ *compound-statement*

where we denote *lambda-parameter-declaration-list*$_{opt}$ as `P`, *exception-specification*$_{opt}$ as `E`, *type-id* or *new-type-id* in *lambda-return-type-clause* `R`, *lambda-local-var-clause*$_{opt}$ as `L`, and *compound-statement* as `B`, is that of

```
<>( P ) E -> R L' B
```

where `L'` is a *lambda-local-var-clause* that contains all *lambda-local-var* entries in `L` and a new entry `&v` for each distinct variable name `v` appearing in `B` and denoting a local object with automatic storage duration in the enclosing scope of the lambda expression. If `B` refers to `this`, or contains member accesses that do not mention `this` but would be converted to class member accesses using `(*this)` according to (9.3.1), `L'` contains `this`.

8 The meaning of a lambda expression

> `<=>(` *lambda-parameter-declaration-list*$_{opt}$ `)` *exception-specification*$_{opt}$ *lambda-return-type-clause* *lambda-local-var-clause*$_{opt}$ *compound-statement*

where we denote *lambda-parameter-declaration-list*$_{opt}$ as `P`, *exception-specification*$_{opt}$ as `E`, *type-id* or *new-type-id* in *lambda-return-type-clause* `R`, *lambda-local-var-clause*$_{opt}$ as `L`, and *compound-statement* as `B`, is that of

```
<>( P ) E -> R L' B
```

where `L'` is a *lambda-local-var-clause* that contains all *lambda-local-var* entries in `L` and a new entry `v` for each distinct variable name `v` appearing in `B` and denoting a local object with automatic storage duration in the enclosing scope of the lambda expression.

9 The meaning of a lambda expression

> *lambda-introducer* `(` *lambda-parameter-declaration-list*$_{opt}$ `)` *exception-specification*$_{opt}$ *lambda-return-type-clause* *lambda-local-var-clause*$_{opt}$ `(` *expression* `)`

where we denote *lambda-introducer* as `I`, *lambda-parameter-declaration-list*$_{opt}$ as `P`, *exception-specification*$_{opt}$ as `E`, *type-id* or *new-type-id* in *lambda-return-type-clause* `R`, *lambda-local-var-clause*$_{opt}$ as `L`, and *expression* with `E1`, is that of

```
I( P ) E -> R L { return E1; }
```

10 The meaning of a lambda expression

> *lambda-introducer* `(` *lambda-parameter-declaration-list*$_{opt}$ `)` *exception-specification*$_{opt}$ *lambda-local-var-clause*$_{opt}$ `(` *expression* `)`

where we denote *lambda-introducer* as `I`, *lambda-parameter-declaration-list*$_{opt}$ as `P`, *exception-specification*$_{opt}$ as `E`, *lambda-local-var-clause*$_{opt}$ as `L`, and *expression* with `E1`, is that of

```
I( P ) E -> decltype(E1) L { return E1; }
```

Chapter 20 General utilities library [utilities]

20.5 Function objects [function.objects]

20.5.17 Class template `nested_function` [func.nest]

```
namespace std {
  template<class> class nested_function; // undefined

  template<class ResType, class... ArgTypes>
  class nested_function<ResType(ArgTypes...)>
  {
  public:
    // 20.5.17.1, trivial members:
    nested_function() = default;
    nested_function(const nested_function&) = default;
    nested_function& operator=(const nested_function&) = default;
    ~nested_function() = default;

    // 20.5.17.2, null values:
    constexpr nested_function(unspecified-null-pointer-type);
    nested_function& operator=(unspecified-null-pointer-type);
    explicit operator bool() const;

    // 20.5.17.3, invocation:
    ResType operator()(ArgTypes...) const;

    // 20.5.17.4, type access:
    typedef ResType result_type;
    typedef T1 argument_type; // iff one argument
    typedef T1 first_argument_type; // iff two arguments
    typedef T2 second_argument_type; // iff two arguments
  };

  // 20.5.17.5, comparisons:
  template <class ResType, class... ArgTypes>
  bool operator==(const nested_function<ResType(ArgTypes...)>&,
                  unspecified-null-pointer-type);
  template <class ResType, class... ArgTypes>
  bool operator==(unspecified-null-pointer-type,
                  const nested_function<ResType(ArgTypes...)>&);
  template <class ResType, class... ArgTypes>
  bool operator!=(const nested_function<ResType(ArgTypes...)>&,
                  unspecified-null-pointer-type);
  template <class ResType, class... ArgTypes>
  bool operator!=(unspecified-null-pointer-type,
                  const nested_function<ResType(ArgTypes...)>&);

} // namespace std
```

1 The `nested_function` class template represents reference-only closures [expr.prim.lambda].

2 A `nested_function` object *`f`* of type `F` is Callable for argument types *`T1, T2, ..., TN`* in *`ArgTypes`* and a return type *`R`*, if, given lvalues *`t1, t2, ..., tN`* of types *`T1, T2, ..., TN`*, respectively, *`INVOKE(f, t1, t2, ..., tN)`* is well-formed (20.5.2) and, if *`R`* is not `void`, convertible to *`ResType`*.

3 The instances of `nested_function` class template are trivial and standard-layout classes (3.9 [basic.types]).

4 Unless otherwise specified, none of the functions in this section throw exceptions.

20.5.17.1 trivial members [func.nest.trivial]

`explicit nested_function()`

1 *Postconditions:* None — the object state is undefined.

`nested_function(const nested_function& f)`

2 *Postconditions:* `*this` is a copy of *`f`*

`nested_function& operator=(const nested_function& f);`

3 *Postconditions:* `*this` is a copy of *`f`*

4 *Returns:* `*this`

`~nested_function();`

5 *Effects:* destroys this

20.5.17.2 null values [func.nest.null]

`nested_function(`*`unspecified-null-pointer-type`*`);`

1 *Postconditions:* `!*this`

`nested_function& operator=(`*`unspecified-null-pointer-type`*`);`

2 *Postconditions:* `!*this`

3 *Returns:* `*this`

`explicit operator bool() const`

4 *Returns:* `true` if `*this` was constructed or copied from a closure, `false` if `*this` was constructed or copied from an *unspecified-null-pointer-type*, undefined otherwise.

5 [*Note:* This conversion can be used in contexts where a bool is expected (e.g., an if condition); however, implicit conversions (e.g., to int) that can occur with bool are not allowed, eliminating some sources of user error. One possible implementation choice for this type is pointer-to-member. — *end note*]

20.5.17.3 invocation [func.nest.invoke]

`ResType` `operator()(`*`ArgTypes`*`... args) const`

1 *Preconditions:* `(bool)*this`

2 *Effects:* Undefined if `*this` was default constructed, constructed from an *unspecified-null-pointer-type* or copied from such. Otherwise, invokes the closure with the given arguments.

3 *Returns:* Nothing if *`ResType`* is `void`, otherwise the return value of the closure.

4 *Throws:* Any exception thrown by the wrapped function object.

20.5.17.4 type access [func.nest.type]

`typedef ResType result_type`

1 *Returns:* The return type of the invocation.

`typedef T1 argument_type`

2 *Returns:* The argument type of the invocation when that invocation takes exactly one argument, otherwise the typedef is not present.

`typedef T1 first_argument_type`

3 *Returns:* The first argument type of the invocation when that invocation takes exactly two arguments, otherwise the typedef is not present.

`typedef T2 second_argument_type`

4 *Returns:* The second argument type of the invocation when that invocation takes exactly two arguments, otherwise the typedef is not present.

20.5.17.5 comparison [func.nest.compare]

```
template <class ResType, class... ArgTypes>
bool operator==(const nested_function<ResType(ArgTypes...)>& f,
                unspecified-null-pointer-type);

template <class ResType, class... ArgTypes>
bool operator==(unspecified-null-pointer-type,
                const nested_function<ResType(ArgTypes...)>& f);
```

1 *Returns:* `!f`

```
template <class ResType, class... ArgTypes>
bool operator!=(const nested_function<ResType(ArgTypes...)>& f,
                unspecified-null-pointer-type);

template <class ResType, class... ArgTypes>
bool operator!=(unspecified-null-pointer-type,
                const nested_function<ResType(ArgTypes...)>& f);
```

2 *Returns:* `(bool)f`