Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4)

Document no: N2550=08-0060

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

2008-02-29

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 2008 Bellevue meeting. The document is a revision of N2529=08-0039 [JFC08], 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.

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].

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:

```
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;
  }
}
```

*jarvi@cs.tamu.edu

```
....
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) { return e.salary() >= min_salary && e.salary() < u_limit; });
```

3 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, Gary Powell, Valentin Samko, Jeremy Siek, Bjarne Stroustrup, Herb Sutter, Jeremiah Willcock, Jon Wray, and Jeffrey Yasskin.

References

[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.

[JFC08] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda expressions and closures: Wording for monomorphic lambdas (revision 3). Technical Report N2529=08-0039, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, February 2008.

[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`.

[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

[EDITORIAL NOTE: Add the following new section:]

5.1.1 Lambda Expressions [expr.prim.lambda]

lambda-expression:
    lambda-introducer lambda-parameter-declaration$_{opt}$ *compound-statement*

lambda-introducer:
    `[` *lambda-capture*$_{opt}$ `]`

lambda-capture:
    capture-default
    capture-list
    capture-default `,` *capture-list*

capture-default:
    `&`
    `=`

capture-list:
    capture
    capture-list `,` *capture*

capture:
    identifier
    `&` *identifier*
    `this`

lambda-parameter-declaration:
    `(` *lambda-parameter-declaration-list*$_{opt}$ `)` *exception-specification*$_{opt}$ *lambda-return-type-clause*$_{opt}$

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

lambda-parameter:
decl-specifier-seq declarator

lambda-return-type-clause:
-> *type-id*

1 The evaluation of a *lambda-expression* results in a *closure object*, which is an rvalue. Invoking the closure object executes the statements specified in the *lambda-expression*'s *compound-statement*. Each lambda expression has a unique type. Except as specified below, the type of the closure object is unspecified. [*Note:* A closure object behaves as a function object ([function.objects], 20.5) whose function call operator, constructors, and data members are defined by the *lambda-expression* and its context. — *end note*]

2 A name in the *lambda-capture* shall be in scope in the context of the lambda expression, and shall be `this` or refer to a local variable or reference with automatic storage duration. [*Note:* A member of an anonymous union is not a variable. — *end note*] The same name shall not appear more than once in a *lambda-capture*. In a *lambda-introducer* of the form [*capture-default* , *capture-list*], if the *capture-default* is `&`, the *capture-list* shall not contain a *capture* having the prefix `&`, otherwise each *capture* in the *capture-list* other than `this` shall have the prefix `&`.

3 An *effective capture set* is defined as follows.

— For a *lambda-introducer* of the form `[]`, the effective capture set is empty.

— For a *lambda-introducer* of the form [*capture-list*], the effective capture set consists of the *capture*s in the *capture-list*.

— For a *lambda-introducer* of the form [*capture-default*] or [*capture-default* , *capture-list*], the effective capture set consists of

— the *capture*s in the *capture-list*, if any; and,

— for each name `v` that appears in the lambda expression and denotes a local variable or reference with automatic storage duration in the context where the lambda expression appears and that does not appear in the *capture-list* or as a parameter name in the *lambda-parameter-declaration-list*, `&v` if the *capture-default* is `&` and `v` otherwise; and

— `this` if the lambda expression contains a member access expression referring to `this` (implicitly or explicitly).

4 The *compound-statement* of a lambda expression shall use ([basic.def.odr], 3.2) an automatic variable or reference from the context where the lambda expression appears only if the name of the variable or reference is a member of the effective capture set, and shall reference `this` (implicitly or explicitly) only if `this` is a member of the effective capture set. The *compound-statement* of a lambda expression shall not refer to a member of an anonymous union with automatic storage duration.

5 A *lambda-expression* defines a function and the *compound-statement* of a *lambda-expression* has an associated function scope ([basic.scope], 3.3).

6 The type of the closure object is a class with a unique name, call it F, considered to be defined at the point where the lambda expression occurs.

Each name `N` in the effective capture set is looked up in the context where the lambda expression appears to determine its object type; in the case of a reference, the object type is the type to which the reference refers. For each element in the effective capture set, F has a private non-static data member as follows:

— if the element is `this`, the data member has some unique name, call it t, and is of the type of `this` ([class.this], 9.3.2);

— if the element is of the form `& N`, the data member has the name `N` and type "reference to object type of `N`";

— otherwise, the element is of the form `N`, the data member has the name `N` and type "cv-unqualified object type of `N`".

The declaration order of the data members is unspecified.

F has a public `const` function call operator ([over.call], 13.5.4) with the following properties:

— The *parameter-declaration-clause* is the *lambda-parameter-declaration-list*.

— The return type is the type denoted by the *type-id* in the *lambda-return-type-clause*; for a lambda expression that does not contain a *lambda-return-type-clause* the return type is `void`, unless the *compound-statement* is of the form `{ return` *expression*`; }`, in which case the return type is the type of *expression*.

— The *exception-specification* is the lambda expression's *exception-specification*, if any.

— The *compound-statement* is obtained from the lambda expression's *compound-statement* as follows: If the lambda expression is within a non-static member function of some class *X*, transform *id-expression*s to class member access syntax as specified in ([class.mfct.non-static], 9.3.1), then replace all occurrences of `this` by *t*. [*Note:* References to captured variables or references within the *compound-statement* refer to the data members of *F*. — *end note*]

7 *F* has an implicitly-declared copy constructor ([class.copy], 12.8), and it has a public move constructor that performs a member-wise move. The copy assignment operator in *F* is defined as deleted. The size of *F* is unspecified.

8 If every name in the effective capture set is preceded by `&`, *F* is publicly derived from `std::reference_closure<R(P)>` (20.5.17), where `R` is the return type and `P` is the *parameter-type-list* of the lambda expression. Converting an object of type *F* to type `std::reference_closure<R(P)>` and invoking its function call operator shall have the same effect as invoking the function call operator of *F*. [*Note:* This requirement effectively means that such *F* must be implemented using a pair of a function pointer and a static scope pointer. — *end note*]

9 The closure object is initialized by direct-initializing each member `N` of *F* with the local variable or reference named `N`; the member *t* is initialized with `this`. If one or more names in the effective capture set are preceded by `&`, the effect of invoking a closure object, or a copy, after the innermost block scope of the context of the lambda expression has been exited is undefined.

5.3.5 Delete [expr.delete]

[EDITORIAL NOTE: Add the following as a new paragraph:]

If a `delete` keyword is immediately followed by empty square brackets, it is interpreted as introducing a delete array expression. [*Note:*It is possible to disambiguate these empty square brackets from a *lambda-introducer* by putting parentheses around the lambda expression. — *end note*]

5.19 Constant expressions [expr.const]

2 [EDITORIAL NOTE: Add the following bullet]

— a *lambda-expression* (5.1.1)

3.3.2 Local scope [basic.scope.local]

2 The potential scope of a function parameter name (including one appearing in a *lambda-parameter-declaration-clause*) or of a function-local predefined variable in a function definition (8.4) begins at its point of declaration. If the function has a function-try-block the potential scope of a parameter or of a function-local predefined variable ends at the end of the last associated handler, otherwise it ends at the end of the outermost block of the function definition. A parameter name shall not be redeclared in the outermost block of the function definition nor in the outermost block of any handler associated with a function-try-block.

Chapter 20 General utilities library [utilities]

20.5 Function objects [function.objects]

2 [EDITORIAL NOTE: Add the following to the synopsis of header <functional>]

```
// 20.5.17, reference_closure
template<class> class reference_closure; // undefined
```

[EDITORIAL NOTE: Add the following new section]

20.5.17 Class template reference_closure [func.referenceclosure]

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

  template<class ResType, class... ArgTypes>
  class reference_closure<ResType(ArgTypes...)>
  {
  public:
    typedef ResType result_type;
    typedef T1 argument_type; // iff sizeof...(ArgTypes) == 1 and ArgTypes contains T1
    typedef T1 first_argument_type; // iff sizeof...(ArgTypes) == 2 and ArgTypes contains T1, T2
    typedef T2 second_argument_type; // iff sizeof...(ArgTypes) == 2 and ArgTypes contains T1, T2

    // 20.5.17.1, trivial members:
    reference_closure() = default;
    reference_closure(const reference_closure&) = default;
    reference_closure& operator=(const reference_closure&) = delete;
    ~reference_closure() = default;

    // 20.5.17.2, null values:
    constexpr reference_closure(nullptr_t);
    reference_closure& operator=(nullptr_t);
    explicit operator bool() const;

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

  // 20.5.17.4, comparisons:
  template <class ResType, class... ArgTypes>
    bool operator==(const reference_closure<ResType(ArgTypes...)>&, nullptr_t);
  template <class ResType, class... ArgTypes>
    bool operator==(nullptr_t, const reference_closure<ResType(ArgTypes...)>&);
  template <class ResType, class... ArgTypes>
    bool operator!=(const reference_closure<ResType(ArgTypes...)>&, nullptr_t);
  template <class ResType, class... ArgTypes>
    bool operator!=(nullptr_t, const reference_closure<ResType(ArgTypes...)>&);

} // namespace std
```

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

2 A `reference_closure` 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 `reference_closure` 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.referenceclosure.trivial]

```
explicit reference_closure()
```

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

```
reference_closure(const reference_closure& f)
```

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

```
~reference_closure();
```

3 *Effects:* destroys this

20.5.17.2 null values [func.referenceclosure.null]

```
reference_closure(nullptr_t);
```

1 *Postconditions:* `!*this`

```
reference_closure& operator=(nullptr_t);
```

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.

20.5.17.3 invocation [func.referenceclosure.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 comparison [func.referenceclosure.compare]

```
template <class ResType, class... ArgTypes>
  bool operator==(const reference_closure<ResType(ArgTypes...)>& f, nullptr_t);

template <class ResType, class... ArgTypes>
  bool operator==(nullptr_t, const reference_closure<ResType(ArgTypes...)>& f);
```

1 *Returns:* `!f`

```
template <class ResType, class... ArgTypes>
  bool operator!=(const reference_closure<ResType(ArgTypes...)>& f, nullptr_t);

template <class ResType, class... ArgTypes>
  bool operator!=(nullptr_t, const reference_closure<ResType(ArgTypes...)>& f);
```

2 *Returns:* `(bool)`*f*