
The Lambda Library: Unnamed Functions in C++

Jaakko Järvi^{1,*}, Gary Powell² and Andrew Lumsdaine¹

¹ *Pervasive Technology Labs, Indiana University, Bloomington, IN 47405, USA*

² *15805 140th CT SE, Renton, WA 98058-7811, USA*

SUMMARY

The Lambda Library (LL) adds a form of *lambda functions* to C++, which are common in functional programming languages. The LL is implemented as a template library using standard C++; thus no language extensions or preprocessing is required. The LL consists of a rich set of tools for defining unnamed functions. Particularly, these unnamed functions work seamlessly with the generic algorithms in the C++ Standard Library. The LL offers significant improvements, in terms of generality and ease of use, compared to the current tools in the C++ Standard Library.

KEY WORDS: C++; generic programming; functional programming

1. Introduction and motivation

The introduction of the *Standard Template Library* (STL) [1, 2], now part of the C++ Standard Library [3], was a small revolution for C++, making commonly needed container classes and algorithms readily available in a highly generic form. STL algorithms are defined in terms of iterators, rather than tied to specific container types. Furthermore, the algorithms parameterize the action performed on the objects to which the iterators refer. Consequently, one of the arguments to an STL algorithm is typically a *function object*. A function object, or *functor*, is simply any object that can be called as a function. This includes function references and pointers but also instances of classes that overload the function call operator (). More often than not, small and simple function objects are needed solely to be passed to an STL algorithm, having no further use in the program. Normal C++ functions or function object classes are not well suited for this purpose; defining them is verbose and adds unnecessary names to

*Correspondence to: Pervasive Technology Labs, Indiana University, Bloomington, IN 47405, USA
E-mail: jajarvi@cs.indiana.edu

the program. What the STL programming style really calls for, is a mechanism for creating unnamed functions that can be defined where they are used, at the call sites of STL algorithms.

The ability to define unnamed functions is a standard feature in functional programming languages, where such functions are referred to as *lambda abstractions*, or *lambda functions*. This feature does not appear in C++, or in other mainstream procedural or object-oriented languages (except for *Eiffel*, where *agents* [4], a recently added language feature, provide means to define unnamed functions). In this paper we describe the *Lambda Library* (LL in the sequel) which fixes this ‘omission’ for C++. The LL is a C++ template library that implements a form of lambda abstractions for C++. The library is designed to work seamlessly with STL algorithms.

It should be noted, that STL does provide a set of tools to let the programmer define function objects on the fly. There are predefined function objects for some common cases (such as `plus`, `less` and `negate`), which can be instantiated directly at the call site of an STL algorithm. Further, there are *binder* templates `bind1st` and `bind2nd` for creating unary function objects from binary function objects by *binding* one of the arguments to a constant value. To make the binder templates more generally applicable, the STL contains *adaptors* for creating bindable function objects from function pointers (`ptr_fun`) and from pointers to member functions (`mem_fun` and `mem_fun_ref`). Finally, some STL implementations contain function composition operations, projections and so forth as extensions to the standard [5].

The goal of all these tools is clear: to make it possible to write unnamed functions in a call to an STL algorithm. However, the current set of tools leaves much room for improvement. Unnamed functors built as compositions of standard function objects, binders, adaptors etc. are hard to read in all but the simplest cases. Moreover, writing ‘lambda abstractions’ with the standard tools is full of restrictions. For example, the standard binders allow only one argument of a binary function to be bound; there are no binders for 3-ary, 4-ary etc. functions. Also, there are technical restrictions that prevent certain kinds of functions with side-effects to be used with binder templates. (See [6, 7] for a more in-depth analysis and discussions about these restrictions.) In the face of these restrictions the programmer must often write explicit, although mechanical, function object classes just to be able to make a single STL algorithm invocation. This is a considerable programming overhead, and not infrequently leads to replacing the invocation with a set of lower level looping constructs, abandoning thus the otherwise intuitive and functional programming style.

The Lambda Library provides solutions to the above problems. The LL syntax for unnamed functions is intuitive and most of the arbitrary restrictions are removed. The concrete consequences of the LL on the tools in the Standard Library are:

- The standard functors `plus`, `minus`, `less` etc. become unnecessary. Instead, the corresponding operators `+`, `-`, `<`, etc. can be used directly.
- The binders `bind1st` and `bind2nd` are replaced by a more general `bind` function template. Using `bind`, arbitrary arguments of practically any C++ function can be bound. Furthermore, `bind` makes `ptr_fun`, `mem_fun` and `mem_fun_ref` adaptors unnecessary.
- No explicit function composition operators are needed.

We show an example to demonstrate what LL’s impact can be on code that uses STL algorithms. The following code is an extract from the documentation of one STL

implementation [5]. Note that in our examples, we do not show the namespace prefixes or using declarations for the `std` namespace, nor for the `boost::lambda` namespace, which is the namespace for the LL definitions.

```
list<int> L;
...
list<int>::iterator new_end =
    remove_if(L.begin(), L.end(),
              compose2(logical_and<bool>(),
                      bind2nd(greater<int>(), 100),
                      bind2nd(less<int>(), 1000)));
L.erase(new_end, L.end());
```

The effect of the code is to remove from the list `L` all elements that are greater than 100 and less than 1000. As a contrast, the same example written with the tools in the LL becomes (repeating only the `remove_if` invocation):

```
list<int>::iterator new_end =
    remove_if(L.begin, L.end(), _1 > 100 && _1 < 1000)
```

The expression `_1 > 100 && _1 < 1000` defines the same unnamed function as the `compose2` call in the first example. Comparing the two versions, in the latter we replace the function objects `greater<int>()` and `less<int>()` with the operators `>` and `<`. Further, the operator `&&` replaces `logical_and<bool>()`, and the two `bind2nd` calls become unnecessary, as well as the call to `compose2`. The variable `_1` is a *placeholder* representing the (first) parameter of the unnamed function. At each iteration, the placeholders will be substituted by the element from the container `L`. Also, with an optimizing compiler, there is no significant performance difference between the two `remove_if` invocations (in fact, the LL version proved to be somewhat faster with one of the compilers we tested, see Section 5, Figure 6).

1.1. Introduction to lambda expressions

The LL can be viewed as an ‘ordinary’ template library, with an interface consisting of a set of function and class templates. Alternatively, the LL can be regarded as a language extension adding lambda functions to C++. In essence, lambda functions are expressions that define unnamed functions. The syntax varies between languages (and between different forms of lambda calculus), but abstractly the basic form of a lambda expressions is

$$\lambda x_1, \dots, x_n. e$$

where x_1, \dots, x_n are the parameters of the function and e is the expression that defines the value of the function in terms of the parameters x_1, \dots, x_n . In the C++ version of lambda expressions the variable declaration part $\lambda x_1, \dots, x_n$ is missing and the formal parameters have predefined names. There are three such predefined formal parameters, called *placeholders*: `_1`, `_2` and `_3`. They refer to the first, second and third argument of the function defined by the lambda expression. For example, the C++ version of the lambda expression

$\lambda xy.x + y$

is

`_1 + _2`

Hence, there is no syntactic keyword for C++ lambda expressions. The use of a placeholder as an operand implies that the operator invocation is a lambda expression. However, this is true only for operator invocations. Lambda expressions containing function calls, control structures, casts etc. require special syntactic constructs. Most importantly, function calls need to be wrapped inside the `bind` function. As an example, consider the lambda expression:

$\lambda xy.foo(x, y)$

Rather than `foo(_1, _2)`, the C++ counterpart for this expression is:

`bind(foo, _1, _2)`

The lambda expression defines a C++ function object, and hence function application syntax is like calling any other function object, for instance: `(_1 + _2)(i, j)`. Regarding terminology, we call, e.g., `_1 + _2` a *lambda expression*. A function object created as a result of evaluating a lambda expression is a *lambda functor*.

2. Basic usage

This section describes the rules for writing lambda expressions consisting of operator invocations and different types of functions and function-like constructs. The basic rules are straightforward, but some operators in C++ have special semantics, which is reflected in the corresponding lambda expressions. Different function types are another source of exceptions to the basic rules. We start with some simple expressions and explain the special cases later in this section. First, we initialize the elements of a container, say, a `list`, to the value 1:

```
list<int> v(10);
for_each(v.begin(), v.end(), _1 = 1);
```

The expression `_1 = 1` creates a lambda functor that assigns the value 1 to every element in `v`; The placeholder `_1` is an empty slot, which will be filled with a value at each iteration. Where a placeholder is used in place of an actual argument, we say that the argument is *open*.

Next, we create a container of pointers and make them point to the elements in the first container `v`:

```
list<int*> vp(10);
transform(v.begin(), v.end(), vp.begin(), &_amp;_1);
```

The expression `&_1` creates a lambda functor for getting the address of each element in `v`. Each address is assigned to the corresponding element in `vp`.

The next code fragment changes the values in `v`. For each element the function `foo` is called. The original value of the element is passed as an argument to `foo`, and the result is assigned back to the element location:

```
int foo(int);
for_each(v.begin(), v.end(), _1 = bind(foo, _1));
```

The next step is to sort the elements of `vp`:

```
sort(vp.begin(), vp.end(), *_1 > *_2);
```

In this call to `sort`, we are sorting the elements by their contents in descending order. Note that the lambda expression `*_1 > *_2` contains two different placeholders, `_1` and `_2`. Consequently, it creates a binary lambda functor. When this functor is called, the first argument is substituted for `_1` and the second argument for `_2`. Finally, the following `for_each` call outputs the sorted content of `vp` separated by line breaks:

```
for_each(vp.begin(), vp.end(), cout << *_1 << '\n');
```

Note that normal (non-lambda) subexpressions are evaluated immediately. This may cause surprises. For instance, if the previous example is rewritten as

```
for_each(vp.begin(), vp.end(), cout << '\n' << *_1);
```

the subexpression `cout << '\n'` is evaluated before the lambda functor is created and the effect is to output a single line break, followed by the elements of `vp`. There is a straightforward way to prevent this from taking place, as explained in Section 2.4.

2.1. About placeholders

As described in Section 1.1, the formal parameters, i.e., the placeholder variables, of the LL lambda expressions have predefined names. So far we have used the placeholders `_1` and `_2`, and mentioned that `_3` exists as well. The use of placeholders in the lambda expression determines the arity of the lambda functor. The highest placeholder index is decisive. For example:

```
_1 + 5           // unary
_1 + _2         // binary
_3 - _2 - _1    // 3-ary
_1 * _1 + _1    // unary
_3 + 10         // 3-ary
```

If the same placeholder occurs multiple times in the same lambda expression, as in the fourth line, the corresponding actual argument is substituted for each occurrence of the placeholder. The last line creates a 3-ary function, which adds 10 to its *third* argument and discards the first two arguments.

Having three placeholders available means that lambda functors can take one, two or three arguments passed in by the STL algorithm; zero parameters is possible as well. It would be straightforward to support higher arities, but no STL algorithm accepts a functor with the number of arguments greater than two, so not more than three placeholders are provided.

The placeholders are variables defined by the LL. The objects themselves are not important but rather their types are. They serve as tags, which allow the placeholders to be recognized

from the arguments stored in a lambda functor, and later replaced with the actual arguments that are passed in when the lambda functor is called. Note that as the placeholders are just ordinary variables, it is easy to define the placeholder names to ones own liking.

Veldhuizen [8] was the first to introduce the concept of using placeholders in expression templates, specifically as index placeholders in vector and matrix expressions. The LL placeholders are different from the original ones, as the types of the arguments that the placeholders stand for do not need to be specified. This means that the lambda functor is polymorphic, and can be called with arguments of any type for which the underlying expression makes sense. Consider again the first example we showed in Section 2:

```
list<int> v(10);
for_each(v.begin(), v.end(), _1 = 1);
```

The lambda expression `_1 = 1` creates a unary lambda functor, which can be called with any object `x`, for which `x = 1` is a valid expression. The `for_each` algorithm iterates over a container of integers, thus the lambda functor is called with an argument of type `int`.

As the types of the actual arguments substituting the placeholder arguments are not known until instantiating a call to the lambda functor, the return type of a lambda functor is in general not known either. Hence, to be able to define the return types of the lambda functors, a mechanism to query a type of an expression is needed—a mechanism that C++ is presently lacking. Consequently, the LL defines a complex set of traits classes (see Section 4.4) that implements typing rules mapping the argument types to the result type of a lambda functor. This type deduction system covers both built-in operators, and user-defined operators that follow normal return type conventions (`bool` for relational operators, the type of the left-hand argument for shifting operators etc.). For user-defined operators which do not follow these conventions, the deduction system is relatively easy to extend.

2.2. About bound arguments

Bound arguments of a lambda expression are stored in the lambda functor object. There are alternative ways of doing this, and the choices have consequences on whether side effects to the bound arguments are allowed or not. Basically the lambda functors can store temporary copies of the arguments, or hold `const` or non-`const` references to them. The default is to store the arguments as `const` copies, which prevents side effects. This means that the value of a bound argument is fixed when the lambda functor is created and remains constant during its lifetime. For example, the result of the lambda functor invocation below is 11, not 20:

```
int i = 1; (_1 + i)(i = 10);
```

In other words, the lambda expression `_1 + i` creates the lambda function $\lambda x.x + 1$ rather than $\lambda x.x + i$.

As said, this is the default, and for some expressions it makes more sense to store the arguments as references and allow side effects to the arguments. We may have arguments that cannot be copied, or that are very expensive to copy. Furthermore, the side effects are sometimes deliberate. As an example, consider the lambda expression `i += _1`. The obvious

intention is that calls to the lambda functor affect the value of the variable `i`, rather than some temporary copy of it. The LL has this behavior: the lambda functors resulting from the compound assignment operators (`+=`, `*=`, etc.) store a non-`const` reference to the left argument. To make the streaming operators (`<<` and `>>`) operate as expected, the stream argument is stored as a reference. Finally, as array types cannot be copied, lambda functors store references to arguments that are of array types.

For all cases, LL provides means for overriding the default storing mechanism. For example, any bound argument in a lambda expression can be wrapped with a function named `ref` to state that the lambda functor should store a reference to the argument. Regarding the preceding example, the lambda expression $\lambda x.x + i$ can be created with the aid of the `ref` function as `_1 + ref(i)`. The function `cref` is analogous, stating that a `const` reference to the argument should be stored. For an in-depth discussion about this issue, see [9].

2.3. Operators as lambda expressions

We have overloaded almost every operator for lambda expressions. Hence, the basic rule is that any operand of any operator can be replaced with a placeholder, or with a lambda expression. All the preceding code examples follow this rule. However, there are some restrictions:

- The return types of operators `->`, `->.`, `new`, `delete`, `new[]` and `delete[]` cannot be chosen freely. Consequently, we cannot overload them for lambda expressions.
- It is not possible to overload the `.`, `.*`, and `?:` operators in C++. For the conditional operator `?:`, the LL provides a wrapper function with the same functionality, but for the other two operators, this is not possible.
- The assignment and subscript operators must be defined as member functions, which creates some asymmetry to lambda expressions. For example:

```
int i;
_1 = 1; // a valid lambda expression
i = _1; // error, no assignment from placeholder type to int
```

A workaround for this situation is explained in Section 2.4.

- As stated in Section 2.1, the return type deduction system may not handle all user-defined operators. For example, the return type of all comparison operators is expected to be `bool`. If this is not true for some user-defined comparison operator, return type deduction fails. In such cases the deduction system can either be extended, or temporarily overridden by explicit type information (see Section 2.5.4).

The LL overloads the comma operator for sequencing lambda expressions together. This idea first appeared in the *Expression Template Library* [10]. Since comma is also the separator between function arguments, extra parentheses are sometimes necessary to write syntactically correct lambda expressions:

```
for_each(c.begin(), c.end(), (cout << _1 << '\n', clog << _1 << '\n'));
```

Here the parenthesis are used to group the two lambda expressions into one expression, as opposed to trying to call the `for_each` function with four arguments. The LL follows the

C++ rule for always evaluating the left-hand operand of a comma expression before the right-hand operand. In the above example, this means that each element of `c` is guaranteed to be first written to `cout` and then to `clog`. Analogously, the short circuiting rules for the operators `&&`, and `||` are respected as well. For example, the following code sets all negative elements of some container `c` to zero:

```
for_each(c.begin(), c.end(), _1 < 0 && _1 = 0);
```

2.4. Delayed constants and variables

It is sometimes necessary to turn a variable or a constant into a lambda functor. We call such lambda functors *delayed variables*, or *delayed constants*, respectively. The need for delayed variables and constants arises when we want to write lambda expressions that are operator invocations, but none of the operands is a placeholder. For example, suppose we wanted to output a space separated list of the elements in some container `c`. Our first attempt might be:

```
for_each(c.begin(), c.end(), cout << " " << _1);
```

However, this piece of code outputs a single space, followed by the elements of `c` without any delimiters. The subexpression `cout << " "` is evaluated first, and it is not a lambda expression. It merely outputs a space and returns a reference to `cout`, rather than creates a lambda functor.

To get the effect we want, the constant `" "` must be turned into a lambda functor with the `constant` function:

```
for_each(c.begin(), c.end(), cout << constant(" ") << _1);
```

Now rather than writing to the stream immediately, the `operator<<` call with `cout` and a lambda functor builds another lambda functor. This lambda functor will be evaluated later at each iteration and we get the desired result.

A delayed variable is simply a lambda functor containing a reference to a regular C++ variable and is created with the function template `var`. A call `var(i)` turns some variable `i` into a lambda expression. A somewhat artificial, but hopefully illustrative, example is to compute the number of elements in a container using the `for_each` algorithm:

```
int count = 0;
for_each(c.begin(), c.end(), var(count)++);
```

The variable `count` is delayed. Hence, the expression `count++` is evaluated at each iteration within the body of the `for_each` function.

A delayed variable, or a constant, can be created outside the lambda expression as well. The template classes `var_type` and `constant_type` serve for this purpose. Using `var_type` the previous example becomes:

```
int count = 0;
var_type<int>::type vcount(var(count));
for_each(c.begin(), c.end(), vcount++);
```

This feature is useful if the same variable appears repeatedly in a lambda expression.

In Section 2.3 we brought up the asymmetry within lambda assignment and subscript operators. As becomes clear from the above examples, delaying the evaluation of a variable with `var` is a solution to this problem:

```
int i;
i = _1;      // error
var(i) = _1; // ok
```

2.5. Functions as lambda expressions

The use of a placeholder as one of the operands turns an operator invocation into a lambda expression implicitly. For ordinary function calls this is not the case. Instead, an explicit syntactic construct is needed, and the `bind` function template serves for this purpose. The syntax of a lambda expression created with the `bind` function is:

```
bind(target-function, bind-argument-list)
```

We use the term *bind expression* to refer to this type of lambda expressions. In a bind expression, the `bind-argument-list` must be a valid argument list for `target-function`, except that any argument can be replaced with a placeholder, or more generally, with another lambda expression.

The target function can be a pointer to function, a reference to function or a function object. Moreover, it can be a pointer to a member function or even a placeholder, or again more generally, a lambda expression. In the last case the result of evaluating the corresponding lambda functor must naturally be a function that can be called with the `bind-argument-list` after substitutions. Note that we use the term *target function* with all types of lambda expressions to denote the underlying operation of the lambda expression. For example, the target function of the lambda expression `_1 + _2` is `operator+`.

2.5.1. Function pointers as targets

The target function can be a pointer or a reference to a non-member function (or to a static member function). For example, suppose `A`, `B`, `C` and `X` are some types:

```
X foo(A, B, C); A a; B b; C c;
...
bind(foo, _1, _2, c)
bind(&foo, _1, _2, c)
bind(_1, a, b, c)
```

The first bind expression returns a binary lambda functor. The second bind expression has an equivalent functionality; it just uses a function pointer instead of a reference. The third bind expression shows the case where the target function is left open; the resulting lambda functor takes one parameter, the function to be called with the arguments `a`, `b` and `c`.

In C++, it is possible to take the address of an overloaded function only if the address is assigned to or used to initialize a properly typed variable. This means that overloaded functions cannot be used in bind expressions directly:

```
void foo(int); void foo(float); int i;
...
bind(&foo, _1); // error
...
void (*pf1)(int) = &foo;
bind(pf1, _1); // ok
...
bind(static_cast<void*(int)>(&foo), _1); // ok as well
```

Another notable limitation is that an uninstantiated template function cannot be used as a target function, as discussed in Section 3.5.

2.5.2. Member functions as targets

By convention, we have chosen to declare the `bind` functions for pointers to member functions with the following format:

```
bind(target-member-function, object-argument, bind-argument-list)
```

If the first argument is a pointer to a member function of some class `A`, the second argument is the *object argument*, that is, an object of type `A` for which the member function is to be called. In fact, a bound object argument can be either a reference or pointer to the object; the LL supports both cases with the same interface:

```
bool A::foo(int) const; A a;
vector<int> ints;
...
find_if(ints.begin(), ints.end(), bind(&A::foo, a, _1)); // reference is ok
find_if(ints.begin(), ints.end(), bind(&A::foo, &a, _1)); // pointer is ok
```

Similarly, if the object argument is left open, the resulting lambda functor can be called both via a pointer or a reference:

```
list<A> refs; list<A*> ptrs;
find_if(refs.begin(), refs.end(), bind(&A::foo, _1, 1));
find_if(ptrs.begin(), ptrs.end(), bind(&A::foo, _1, 1));
```

Even though the interfaces are the same, there are important semantic differences between using a pointer or a reference as the object argument. The differences stem from the way `bind` functions take their parameters, and how the bound parameters are stored within the lambda functor. The object argument has the same storing mechanism than any other bind argument slot (see Section 2.2); it is stored as a `const` copy in the lambda functor. This creates some asymmetry between the lambda functor and the original member function, and between seemingly similar lambda functors. For example:

```
class B {
    int i; mutable int j;
public:
    B(int ii, int jj) : i(ii), j(jj) {};
    void set_i(int x) { i = x; };
    void set_j(int x) const { j = x; };
};
```

When a pointer is used, the behavior is what the programmer might expect:

```
B b(0,0); int k = 1;
bind(&B::set_i, &b, _1)(k); // b.i == 1
bind(&B::set_j, &b, _1)(k); // b.j == 1
```

Even though a `const` copy of the object argument is stored, the original object `b` is still modified. This is because the object argument is a pointer, and the pointer is copied, rather than the object to which it points.

When we use a reference, the behavior is different:

```
B b(0,0); int k = 1;
bind(&B::set_i, b, _1)(k); // error; a const copy of b is stored.
                          // Cannot call a non-const function set_i
bind(&B::set_j, b, _1)(k); // b.j == 0, as a copy of b is modified
```

To prevent the copying from taking place, one can use the `ref` or `cref` wrappers (see Section 2.2):

```
bind(&B::set_i, ref(b), _1)(k); // b.i == 1
bind(&B::set_j, cref(b), _1)(k); // b.j == 1
```

Note that the preceding discussion is relevant only for bound arguments. If the object argument is open, the parameter passing mode is always by reference and no copying takes place.

Even though the above rules may seem arbitrary, the LL in fact treats each argument position in a `bind` expression similarly. Viewing member functions as ordinary functions with the object as an additional argument to the function helps in understanding the semantics of `bind` expressions.

2.5.3. Function objects as targets

Function objects can also be used as target functions. The library cannot, however, know the return type of an arbitrary function object class. In the STL, *adaptable* function object classes are required to define the `result_type` typedef. Earlier versions of the LL used this same convention to ‘export’ the return type out of the function object class. However, this method can handle only one return type per function object class. Consequently, if a class overloads the function call operator, all the overloaded definitions must have the same return type. Moreover, if the function call operator is a template, the result type may depend on the

template parameters. Hence, the typedef ought to be a template too, which the C++ language does not presently support.

To overcome this restriction, the LL uses a different method for specifying the return type(s) of the function call operator(s), which is slightly more complex, but also more expressive. Instead of a typedef, the function object class defines a class template which serves as a metafunction that maps the argument types to the result type. The LL borrows this idea from the FC++ library [11, 12] (see Section 5.1). The return type deduction system in FC++ is entirely based on the requirement that each function object defines its return type as a typedef inside a nested template class.

As an example, the return type of the function call operator in the `functor` class below is dependent on the type of the third argument:

```
struct functor {
    template<class T1, class T2, class T3>
    T3 operator()(const T1& t1, const T2& t2, const T3& t3);

    template <class Tuple>
    struct sig { typedef typename element<3, Tuple>::type type; };
};
```

The `sig` template with the typedef `type` inside of it make this information available to the LL. Having the `sig` template defined, instances of `functor` can be used within `bind` expressions just as objects of built-in function types:

```
functor f;
A a; B b; C c;
list<functor> f_list; list<C> c_list;
...
for_each(c_list.begin(), c_list.end(), bind(f, a, b, _1));
for_each(f_list.begin(), f_list.end(), bind(_1, a, b, c));
```

Prior to invoking the function call operator with some arguments, the LL bundles the types of these arguments into a *tuple* type (see page 19) and instantiates the `sig` template with this tuple type as the sole template argument. The `sig` template now has the types of arguments to the function call operator accessible in its template argument, and can use this information in defining the return type. In the above example, the return type is the same as the type of the third actual argument. The expression `typename element<3, Tuple>::type` extracts this type from the `Tuple` template argument. In general, the `sig` template can define an arbitrary type function.

Note that if an argument to the function call operator has a `const` or `volatile` qualifier (commonly referred to as *cv-qualifiers*), or both, the corresponding type in the argument tuple has the same set of qualifiers. We retain the qualifiers as their absence or presence may have an effect on the return type. Furthermore, the function object class can define `const` (or `volatile`) versions of the function call operators which may have different return types. To account for this, the appropriately cv-qualified function object type is included in the argument type tuple

as the first element (at index 0). These features are refinements over the `sig` templates of the FC++ library, and are needed to be able to cover the return types of an arbitrary set of templated and overloaded function call operators.

Finally, there should not be many reasons to continue to use the standard functors as the target functions in `bind` expressions, but the LL nevertheless provides a wrapper function `std_functor` that adds a `sig` template for a standard function. The wrapper works with any function object class that uses the STL convention to define the return type as the `result_type` typedef. For example:

```
bind(plus<int>(), 1, 2)(); // error, no sig template
bind(std_functor(plus<int>(), 1, 2)()); // ok
```

2.5.4. Overriding the deduced return type

If the `sig` template is not defined the return type deduction system cannot deduce the return type of a function object. This can also be the case with user defined operators. Hence, there is a special lambda expression with which the return type can be written explicitly, overriding the deduction system. To state that the return type of the lambda functor defined by the lambda expression `e` is `T`, one can write `ret<T>(e)`. For example:

```
A a; B b;
C operator+(A, B);
...
(_1 + _2)(a, b); // error, the LL does not know the return type
ret<C>(_1 + _2)(a, b); // ok
```

Obviously `T` cannot be an arbitrary type; the true result of the lambda functor must be implicitly convertible to `T`. For `bind` expressions, there is a short-hand notation that can be used instead of `ret`. E.g., `ret<T>(bind(f, _1))` can be written as `bind<T>(f, _1)`. An alternative for `ret` is to extend the return type deduction templates to cover the user defined types, described in Section 3.6.

3. Advanced features

Our goal has been to make the LL as complete as possible in the sense that any C++ expression could be turned into a lambda expression. This section describes how to write control structures as lambda expressions, how to construct and destruct objects in lambda expressions and even how to do exception handling in lambda expressions.

3.1. Control lambda expressions

Control lambda expressions create lambda functors that implement the behavior of some control structure. The idea of providing such lambda expressions originates from the Expression Template Library [10]. The control lambda expressions that LL provides are

`if_then`, `if_then_else`, `for_loop`, `while_loop`, `do_while_loop`, and `switch_statement`. The arguments to these function templates are lambda functors. For example, the following code outputs all even elements of some container `a`:

```
for_each(a.begin(), a.end(), if_then(_1 % 2 == 0, cout << _1));
```

As an example of a loop control lambda expression, the pseudo code definition of `for_loop` is:

```
for_loop(init, test, increment, body)
```

where all the arguments are lambda functors. As a concrete example, the following code adds 6 to each element of a two-dimensional array:

```
int a[5][10]; int i;
for_each(a, a+5,
         for_loop(var(i) = 0, var(i) < 10, ++var(i), _1[var(i)] += 6));
```

Note the use of delayed variables to turn the arguments of `for_loop` into lambda expressions. As stated in Section 2.4, we can avoid the repeated wrapping of a variable with `var` if we create the delayed variable beforehand using the `var_type` template. Using `var_type` the above example becomes:

```
int i;
var_type<int>::type vi(var(i));
for_each(a, a+5, for_loop(vi = 0, vi < 10, ++vi, _1[vi] += 6));
```

Other loop structures are analogous to `for_loop`. The return type of all control lambda functors is `void`.

The lambda expressions for `switch` control structures are more complex since the number of cases may vary. The general form of a switch lambda expression is:

```
switch_statement(condition,
                 case_statement<label>(lambda expression),
                 case_statement<label>(lambda expression),
                 ...
                 default_statement(lambda expression)
)
```

The *condition* argument must be a lambda expression that creates a lambda functor with an integral return type. The different cases are created with the `case_statement` functions, and the optional default case with the `default_statement` function. The case labels are given as explicitly specified template arguments to `case_statement` functions and the `break` statements are implicitly part of each case. For example, `case_statement<1>(a)`, where `a` is some lambda functor, generates the code:

```
case 1:
    evaluate lambda functor a;
    break;
```

We have specialized the `switch_statement` function for up to 9 case statements. As a concrete example, the following code iterates over some container `v` of integral elements and outputs `zero` for each 0, `one` for each 1, and `other: n` for any other value `n`. Note that another lambda expression is sequenced after the `switch_statement` to output a line break after each element:

```
std::for_each(v.begin(), v.end(),
  ( switch_statement(_1,
    case_statement<0>(std::cout << constant("zero")),
    case_statement<1>(std::cout << constant("one")),
    default_statement(cout << constant("other: ") << _1)),
    cout << constant("\n") )
);
```

3.2. Alternative syntax for control expressions

We have opted to use the normal function call syntax to build the control constructs. This is however not the only possibility. Using operator overloading and choosing member variable names suitably, we can build a syntax closer to the syntax of the built-in control structures. For example, with this syntax the lambda expression:

```
if_then_else(cond, then_part, else_part);
```

is written as:

```
if_(cond)[then_part].else_[else_part];
```

This syntax, and the mechanism for adding it into the framework of the LL is work by Joel de Guzman [13].

3.3. Constructors and destructors as lambda expressions

Operators `new` and `delete` can be overloaded, but their return types are fixed. Particularly, the return types cannot be lambda functors. Furthermore, it is not possible to take the address of a constructor or destructor, preventing their use as target functions in bind expressions. To circumvent the above restrictions, the LL provides wrapper classes for these operations. Instances of these classes are function objects—and can thus be used as target functions. For example:

```
int* a[10];
for_each(a, a+10, _1 = bind(new_ptr<int>()));
for_each(a, a+10, bind(delete_ptr(), _1));
```

The template class `new_ptr` is a wrapper for a `new` invocation. The type of the object to be constructed is given as a template argument. Note that `new_ptr` can take arguments as well. They are passed directly to the constructor invocation and thus allow calls to constructors which take arguments. Similarly, the `delete_ptr` is a wrapper class for the `delete` operator, deleting its argument when invoked. We have also defined `new_array` and `delete_array` for `new[]` and `delete[]`. The constructor wrappers are created as:

```
constructor<T>(args)
```

where the wrapped call is: `T(args)`. The complementary function object class `destructor` exists as well. The following example reads integers from two containers (`x` and `y`), constructs pairs out of them, and inserts the pairs into a third container:

```
vector<pair<int, int> > v;
transform(x.begin(), x.end(), y.begin(), back_inserter(v),
          bind(constructor<pair<int, int> >(), _1, _2));
```

3.4. Exception handling in lambda expressions

The LL provides lambda expressions for throwing and catching exceptions. The form of a lambda expression for try catch blocks is as follows:

```
try_catch(
    lambda expression,
    catch_exception<type>(lambda expression),
    catch_exception<type>(lambda expression),
    ...
    catch_all(lambda expression)
)
```

The first lambda expression is the try block. Each `catch_exception` defines a catch block; the type of the exception to catch is specified with the explicit template argument. The *lambda expression* within the `catch_exception` defines the actions to take if the exception is caught. The last catch block can alternatively be a call to `catch_exception<type>` or to `catch_all`. We have used `catch_all` to mean `catch(...)`, since it is not possible to write `catch_exception<...>`.

Lambda functors for throwing exceptions are created with the unary function `throw_exception`. The argument to this function is the exception to be thrown, or a lambda functor which creates the exception to be thrown. A lambda functor for rethrowing exceptions is created with the nullary `rethrow` function.

Figure 1 demonstrates the use of the LL exception handling tools. The first catch block is for handling exceptions of type `foo_ex`. Note the use of the `_1` placeholder in the lambda expression that defines the body of the handler.

The second handler catches exceptions from the Standard Library, writes an informative message to the `cout` stream and constructs and throws an exception of another type, `bar_ex`. An object of type `std::exception` carries a string explaining the cause of the exception. This explanation can be queried with the zero-argument `what` member function; `bind(&std::exception::what, _e)` is the lambda expression for creating the lambda functor for calling `what`. Note the use of `_e` as the argument. It is a special placeholder, which refers to the caught exception object within the handler body. The last handler (`catch_all`) demonstrates rethrowing exceptions.

```

for_each(a.begin(), a.end(),
  try_catch(
    bind(foo, _1), // foo may throw
    catch_exception<foo_ex>(
      cout << constant("foo_ex: ") << "foo argument = " << _1
    ),
    catch_exception<std::exception>(
      cout << constant("exception: ") << bind(&std::exception::what, _e),
      throw_exception(bind(constructor<bar_ex>(), _1))
    ),
    catch_all((cout << constant("Unknown"), rethrow()))
  )
);

```

Figure 1. Throwing and handling exceptions.

3.5. Nesting STL algorithms

In Section 3.1 we showed an example using `for_loop` as the function object passed to the `for_each` algorithm. We repeat the example here:

```

int a[5][10]; int i;
for_each(a, a + 5,
  for_loop(var(i) = 0, var(i) < 10, ++var(i), _1[var(i)] += 6));

```

We could do better: the inner loop should really be another `for_each` invocation. However, `for_each` is a function template, and it is not possible to take an address of an uninstantiated function template. This means, that `for_each` cannot be a target function in a `bind` expression, unless we explicitly instantiate it, which is very tedious. To circumvent this, we use the same approach as with e.g. constructors; provide wrapper function object classes. The LL has a subnamespace `ll`, where the names of the standard algorithms refer to our own function object classes. These function object classes are wrappers for corresponding standard algorithms, and can be used in `bind` expressions. Using a nested `for_each`, the above example becomes:

```

for_each(a, a + 5, bind(ll::for_each(), _1, _1 + 10, _1 += 6));

```

But now we have a problem. When the lambda functor is called from within the outermost `for_each` algorithm, all the occurrences of the `_1` placeholder are replaced by the actual argument, which is a pointer to the beginning a row in the two-dimensional array. But this is nonsensical for the subexpression `_1 += 6`, which should define the action to be performed within the innermost loop. Hence, we want to prevent the argument substitution for this subexpression, and pass the subexpression as such to the inner `for_each`. The LL provides the special function `protect` for just this purpose. It creates a lambda functor that encloses

another lambda functor. When invoked, the outer lambda functor, i.e. the result of `protect`, merely returns the enclosed lambda functor. Using `protect`, our example is written as:

```
for_each(a, a + 5, bind(1l::for_each(), _1, _1 + 10, protect(_1 += 6)));
```

The `protect` function can be applied to each layer of nested lambda functors as needed.

3.6. Extending return type deduction system

In this section, we explain how to extend the return type deduction system to cover user defined operators. In many cases this is not necessary, as the LL defines default return types for operators. For example, the default return type for all comparison operators is `bool`, and as long as the user defined comparison operators have `bool` as their return type, there is no need to write new specializations for the return type deduction classes. Sometimes this cannot be avoided, though.

The overloadable user defined operators are either unary or binary. For both of these arities, there are separate traits templates that define the return types of the different operators. Hence, extending the return type deduction system means simply providing more specializations for these templates. The specializations are of the form `plain_return_type_1<Action, A>` for unary operators, and respectively `plain_return_type_2<Action, A, B>` for binary operators.

The first parameter (`Action`) to all these templates is the *action class*, which specifies the operator. Operators with similar return type rules are grouped together into *action groups*, and only the action class and action group together define the operator unambiguously. As an example, the template instance `arithmetic_action<plus_action>` stands for `operator+`.

The latter parameters, `A` in the unary case, or `A` and `B` in the binary case, stand for the argument types of the operator call. The parameter types are always provided as non-reference types, and do not have `const` or `volatile` qualifiers, which makes specializing easy, as one specialization for each user defined operator, or operator group, is enough. As a side note, the library provides another layer of templates for the cases where a particular operator is overloaded for different cv-qualifications of the same argument types, and the return types of these overloaded versions differ. This layer is described in the library documentation [14].

Suppose the user has overloaded the following operators for some user defined types `X`, `Y` and `Z`:

```
Z operator+(const X&, const Y&);
Z operator-(const X&, const Y&);
```

Now, one can add a specialization stating, that if the left hand argument is of type `X`, and the right hand one of type `Y`, the return type of all such binary arithmetic operators is `Z`:

```
template<class Act>
struct plain_return_type_2<arithmetic_action<Act>, X, Y> {
    typedef Z type;
};
```

Having this specialization defined, the LL can correctly deduce the return type of the above two operators. It is possible to specialize on the level of an individual operator as well, if, say, plus and minus operators had different return types.

4. Implementation of the LL

The LL implementation is based on *expression templates* introduced in [8] (see also [15, Chapter 10] [16]). The basic idea behind expression templates is to overload operators to create *expression objects* to represent the expression and its arguments instead of evaluating the operator instantly. As a result, the type of an expression object can describe a parse tree of the underlying expression. This expression object can be manipulated in various ways (also at compile time) prior to actually evaluating it, for example to yield better performance by preventing the creation of unnecessary temporary objects [8]. In the LL case, this manipulation means substituting the actual arguments for the placeholder objects.

The LL attempts to cover a fairly complete set of expression types. This means that there are many sources of variability: the arity of the lambda functors, the arity of the target functions, the call syntax of the target functions (member functions, non-member functions, operators), boundedness of arguments etc. To be able to cope with the variability with a linear, rather than a combinatorial number of template definitions, the LL consists of several layers. Each layer implements a certain task orthogonal to the tasks of the other layers. Within each layer, it is enough to write template specializations or function overloads with respect to a single varying factor.

Even with the strict layering, the architecture of the LL is somewhat complex. We will not try to describe the library code entirely, but rather show the basic ideas behind the implementation. We do this by looking at the chain of template instantiations taking place while compiling a simple lambda expression `_1 < 0`. This example expression does not use all layers of the library, as operator expressions are a slightly special case. Also, we make several simplifications to the presented code for the sake of clarity. At the end of this section we explain what the differences to the full working implementation are.

To start with, we describe *tuple* types, which were shortly mentioned in Section 2.5.3, along the discussion of the `sig` templates. Tuples are used extensively in the library internally, making the implementation more concise; argument lists can be represented as tuples, and thus treated as a single entity regardless of the true number of arguments, and regardless of whether they are bound or open. Obviously, at some point we must break up the tuples and repeat some code for different argument list lengths in order to be able to call the underlying target functions. However, with the use of tuples this code repetition can be kept to a minimum.

In short, the tuple template is a generalization of the standard `pair` template. An arbitrary number of objects of arbitrary types can be grouped into a tuple. For example, the following definitions are valid tuple types (assuming that `A`, `B` and `C` are valid types):

```
tuple<int, const float, const double*>
tuple<int (*) (int, double), tuple<A, B>, const char&, C&>
```

Just as there is a `make_pair` function for constructing `std::pair` objects, tuples can be created with `make_tuple` functions, or by calling the tuple constructor directly with the elements to be stored. The elements, or element types, can be accessed easily with the expressions:

```
get<N>(aTuple);           // a reference to the Nth element of aTuple
typename element<N, T>::type; // the type of the Nth element of the tuple type T
```

The element index must be known at compile time. The accessor functions incur no runtime overhead. For an interested reader, tuples are described in greater depth in [17] and in the documentation of the *Boost Tuple Library* [18]. Note that as tuples are not built-in C++ types, there is an implementation defined upper limit for the number of arguments, which is currently 10. When we mention a limit on the number of cases in the LL as being 10 or less, this is where the limitation originates.

4.1. Functions to create lambda functors

The central template class in the implementation is `lambda_functor`. It is our expression template class, which is a common umbrella for all different types of lambda functors. Placeholder types, the types of the results of different lambda expressions, such as `_1 + 1` and `bind(foo, 1, _2, _1)`, exception handling lambda functors etc. are all instances of the `lambda_functor` template. This makes it possible to freely mix different sorts of lambda expressions.

We start our walk through the layers with the functions that construct lambda functors. Such functions exist for unary and binary operators, bind expressions, control structures, delayed variables etc. Taking a closer look at the binary operators, there are three overloaded definitions for each operator @:

```
operator@(lambda_functor, any_type)
operator@(any_type, lambda_functor)
operator@(lambda_functor, lambda_functor)
```

When the compiler encounters our example expression `_1 < 0`, the version for `operator<(lambda_functor, any_type)` is selected as a result of the overload resolution. The Figure 2. shows its definition. This operator takes a `lambda_functor` as the left-hand argument and constructs another `lambda_functor`. The arguments `_1` and `0` are stored in the lambda functor, grouped in a tuple object. The template parameters `Arg` and `B` get the values `placeholder<FIRST>` and `int`, and the type of the constructed lambda functor becomes:

```
lambda_functor<
  lambda_functor_base<
    relational_action<less_action>,
    tuple<lambda_functor<placeholder<FIRST> >, const int>
  >
>
```

This type carries information about the invoked operator and the types of the arguments to the operator. The type `relational_action<less_action> >` means that the lambda functor

```

template<class Arg, class B>
inline const
  lambda_functor<
    lambda_functor_base<
      relational_action<less_action>,
      tuple<lambda_functor<Arg>, const B>
    >
  >
operator< (const lambda_functor<Arg>& a, const B& b)
{
  return lambda_functor_base<
    relational_action<less_action>,
    tuple<lambda_functor<Arg>, const B>
  > ( make_tuple(a, b) );
}

```

Figure 2. One of the three `operator<` overloads for lambda functors.

stands for the binary less than operator. The arguments are stored in the lambda functor as an object of type:

```
tuple<lambda_functor<placeholder<FIRST> >, const int>
```

The type of the `_1` placeholder is `lambda_functor<placeholder<FIRST> >`, and 0 is stored as type `const int`. Hence, placeholders are just instances of a templated class `placeholder` wrapped in a `lambda_functor`. We have defined `placeholder<FIRST>` to be the placeholder for the first argument to the lambda functor, `placeholder<SECOND>` the second argument and so forth. The `FIRST`, `SECOND`, etc. are integral constants defined by the library.

In sum, the task of the creation functions is to bundle the target function together with its arguments into a lambda functor object. This is closely related to creating a *closure* of a function.

4.2. The `lambda_functor` template

The `lambda_functor` template defines the function call operators that are called from STL algorithms. The operators are defined for all supported arities (0–3 in the current implementation). In Figure 3, we show the one and two argument cases. Since our example lambda expression has one open parameter, the unary `operator()` gets instantiated when the lambda function is called.

As we explained in Section 2.1, a lambda functor can be called with arguments of any types, provided that this results in a valid call to the underlying function after the argument

```
template <class T>
class lambda_functor : public T {
public:

    typedef T inherited;
    lambda_functor(const T& t) : inherited(t) {}

    // ... nullary case

    template<class A>
    typename inherited::template sig<tuple<A&> >::type
    operator()(A& a) const {
        return inherited::call<
            typename inherited::template sig<tuple<A&> >::type
            >(a, cnull_type(), cnull_type());
    }

    template<class A, class B>
    typename inherited::template sig<tuple<A&, B&> >::type
    operator()(A& a, B& b) const {
        return inherited::call<
            typename inherited::template sig<tuple<A&, B&> >::type
            >(a, b, cnull_type());
    }

    // ... 3 and more arguments
};
```

Figure 3. Part of the `lambda_functor` template. The `template` keyword before the `call` invocations and the `sig` templates is required by the C++ standard; it helps the parser to figure out that `call` and `sig` are templated members rather than member variables.

substitutions have been performed. This is possible since lambda functor templates define the `operator()` functions as member templates.

The task of the `operator()` function template is to initiate the return type deduction chain and forward the call. Both of these tasks are delegated to the base class, which is the sole template parameter of `lambda_functor`. Hence, the `lambda_functor` serves as an interface that relies on its template parameter to be able to do the actual job. This powerful technique, known as *parameterized inheritance* (see e.g. [15, p. 221]), leaves the `lambda_functor` template ‘open-ended’, making it possible to plug in any class as long as it defines the function and the return type deduction class that are accessed from within the `lambda_functor` template. The

fact that we are using inheritance is not crucial here, we could just as well store an object of the type of the template parameter as a member variable. The flexibility comes from delegating the work to whatever class `lambda_functor` is instantiated with. The adoption of this technique is a fairly recent change in the library, suggested by Guzman [13], making it easy to extend the library with new functionality.

4.3. The base class of lambda functor

To qualify as the template parameter, that is, as the base class of `lambda_functor`, a class must define:

- a three argument function `call` (three is the maximum arity of lambda functors). Note that when this function is called from the `lambda_functor`'s function call operators which have lower arities, the missing arguments are filled with special *null_type*-objects (`cnull_type()` gives such an object). This arrangement saves us from providing a `call` function for the lower arities. Note that the null objects do not have any effect on the program execution, as they will eventually be discarded and a decent compiler can optimize them out of existence altogether. Another requirement of the `call` function is, that it must be callable with at least one explicitly specified template parameter, which defines the return type of the function. The reason for using a template argument to specify the return type is an attempt to save the compiler some work. Deducing the return type can be a costly operator for the compiler and since the caller of this function (one of the function call operators in `lambda_functor`) must perform this task anyway, we pass the return type into the function as an explicitly specified template parameter.
- a return type deduction template named `sig` that defines a typedef named `type`. When invoking the `call` function, the LL will use this as the return type of `call` and it must be compatible with the type of the expression `call` really returns, that is, the type of the expression in the return statement in the body of `call`. The parameter of the `sig` template is the tuple type representing the argument list of the `call` function invocation. Hence, `sig` is a type function mapping the arguments types to the return type. The `sig` template here is analogous to the `sig` template in LL aware function objects (see Section 2.5.3) but there are some differences. The first element in the tuple argument of the function object `sig` template is the potentially cv-qualified function object class itself, whereas here the tuple only contains the argument types. This is since whether the lambda functor type is cv-qualified or not, does not have any significance in deducing the return type here. Another difference is in the argument types themselves. For the function object `sig` templates the library guarantees that the argument tuple only contains non-reference types but here the element types can be reference types too; in some cases the return type will be different for reference and non-reference types (cf. built-in comma operator).

Referring again to the example, suppose that the unary function call operator in Figure 3 is called with an argument of type `int`. For example:

```
int i; ... ; (_1 < 0)(i);
```

The unary function call operator in `lambda_func` will be instantiated. The expression for the return type becomes:

```
inherited::sig<tuple<int&> >::type
```

We discuss the return type deduction traits later in Section 4.4; assume for now that the return type is correctly deduced to `bool`. After this deduction, the `call` invocation becomes:

```
inherited::call<bool>(i, cnull_type(), cnull_type())
```

As described above, any class that meets the requirements, i.e., that provides the `call` function and the `sig` template, qualifies as a base class of `lambda_func`. We have chosen to implement most of these base classes as specializations of the `lambda_func_base` template. This makes the purpose of the class immediately clear, but there is also a historical reason for this, as prior to using parameterized inheritance, the `lambda_func` class explicitly inherited from an instance of a template of this name.

The `lambda_func_base` templates take two arguments, the action class and the tuple consisting of the arguments. Our example target function, `operator<`, leads to the instantiation of the specialization shown in Figure 4. We first discuss the `call` function. The parameter types are templated and their types will be automatically deduced. As explained above, we do not need to care about the return type, as it is given as the explicitly specified template parameter at the call site in the `lambda_func` template.

In our example case, when invoked from the `lambda_func::operator()` function, the prototype of the `call` functor becomes:

```
bool call(int& a, const null_type& b, const null_type& c)
```

This function delegates the argument substitution to the `select` functions and calls the `operator<` with the substituted arguments.

The `select` functions perform the selection between bound arguments and placeholders (or other lambda functors). The call

```
select(get<0>(args), a, b, c)
```

selects either the bound argument from the `args` tuple (accessed with `get<0>(args)`) or one of the arguments `a`, `b` or `c`. Which it does, depends on the type of the bound argument. The default case is that we do not know anything about the type. This means that we have a bound argument, and we return that argument as such:

```
template<class Any, class A, class B, class C>
inline Any& select(Any& any, A&, B&, C&) { return any; }
```

However, the stored argument can be a placeholder, in which case one of the other arguments should be chosen. Moreover, the argument could be another lambda functor, in which case that lambda functor should be evaluated with the arguments `a`, `b` and `c`, in effect recursively restarting the evaluation chain. In earlier versions of the library, we used to provide overloaded versions of `select` functions for placeholders. We have since made placeholders full-fledged lambda functors and can therefore treat them as any other lambda functors. Consequently,

```

template<class Args>
class lambda_functor_base<relational_action<less_action>, Args> {
public:
    Args args;
    explicit lambda_functor_base(const Args& a) : args(a) {}

    template<class RET, class A, class B, class C>
    RET call(A& a, B& b, C& c) const {
        return select(get<0>(args), a, b, c) < select(get<1>(args), a, b, c);
    }

    template<class Tuple> class sig {
    private:
        typedef typename arg_t<typename element<0, Args>::type, Tuple>::type X;
        typedef typename arg_t<typename element<1, Args>::type, Tuple>::type Y;
    public:
        typedef typename
            return_type_2<relational_action<less_action>, X, Y>::type type;
    };
};

```

Figure 4. The `lambda_functor_base` specialization for the less than operator.

if the first argument of the `select` function is a lambda functor, we can merely forward the arguments to it and let it decide what to do with them. Also, we can use its `sig` template to deduce the return type:

```

template<class Arg, class A, class B, class C>
inline typename Arg::template sig<tuple<A&, B&, C&> >::type
select (const lambda_functor<Arg>& op, A& a, B& b, C& c) {
    return op(a, b, c);
}

```

In our example, the `select` call in the left hand operand of the less than operator invocation matches the lambda functor case, as the first element in the argument tuple is a placeholder. After the template parameter deduction the function becomes:

```

inline int&
select(const lambda_functor<placeholder<FIRST> >& op,
        int& a, const null_type& b, const null_type& c) {
    return op(a, b, c);
}

```

This is a call to the function call operator of the lambda functor, which will forward the arguments to the `call` function in its base class. The base class `placeholder<FIRST>` is defined as follows.

```
template<> struct placeholder<FIRST> {
    template<class Tuple> struct sig {
        typedef typename element<0, Tuple>::type type;
    };

    template<class RET, class A, class B, class C>
    RET call(A& a, B& b, C& c) const { return a; }
};
```

The `call` function here merely returns its first argument as such, performing the substitution of the first actual argument for the `_1` placeholder. Similarly, the type deduction in the `sig` template selects the first argument in the argument tuple, which equals `A&` in the `call` function.

The second element in the argument tuple has the type `const int` and is not a lambda functor. Hence, the first `select` function matches and returns the stored argument as such.

4.4. Return type deduction

When the compiler has deduced the argument types of the lambda functor's function call operator, the return type templates are instantiated to resolve the return type of this operator. This task is analogous to the argument substitution. However, now we operate on types instead of objects. As input, the return type deduction system takes the lambda functor type, together with the types of the arguments it was called with.

In order to determine the argument types, the compiler may have to resort to the return type deduction system recursively. For example, consider the lambda expression, `_1 + (_1 * _2)`. Suppose this binary function is called with objects of type `A` and `B`. Before the compiler knows the second argument type for the `operator+`, it must perform the return type deduction for multiplication of types `A` and `B`. Say this results in a type `C`. Then the compiler knows that the addition is for types `A` and `C`, and can proceed with the type deduction for that operation.

Consider again the `sig` template in the `lambda_functor_base` class for the less than operator:

```
template<class Tuple> class sig {
private:
    typedef typename arg_t<typename element<0, Args>::type, Tuple>::type X;
    typedef typename arg_t<typename element<1, Args>::type, Tuple>::type Y;
public:
    typedef typename
        return_type_2<relational_action<less_action>, X, Y>::type type;
};
```

In most cases the return type of the the less than operator is `bool` but it would be too hasty to encode the `sig` class as:

```
template<class Tuple> struct sig { typedef bool type; }
```

Instead, we need to first determine the return types of the arguments, much of the same way as we did with the `select` functions. The definition of the `sig` class can be interpreted as follows: Use the traits class `return_type_2<relational_action<less_action>, X, Y>` to deduce the return type of calling the less than operator with some parameters of types `X` and `Y`. The parameter `X` is obtained by recursively performing the return type deduction for the first stored argument in the lambda functor, and `Y` respectively for the second argument. The expression `element<N, Args>::type` resolves to the `N`th type in the `Args` tuple.

We can define the `arg_t` template as follows:

```
template <class Any, class Args> struct arg_t { typedef Any type; };

template arg_t<lambda_functor<T>, Args> {
    typename T::template sig<Args>::type type;
};
```

For lambda functors, we query the return type from the `sig` template of its base class. For all other types, we have a bound argument, and use its type as such. In our example case, the latter definition is instantiated to deduce the first argument type `X` as:

```
placeholder<FIRST>::sig<tuple<int&& >>::type
```

which equals `int&`. The second argument type `Y` becomes `const int` by the first `arg_t` definition.

The `return_type_2` template has specializations for different (binary) actions or action groups. All relational operators are defined to return `bool`:

```
template<class A, class B, class Act>
struct return_type_2<relational_action<Act>, A, B> { typedef bool type; };
```

4.5. Summary of the example

We have now covered the following function call chain:

1. The overloaded `operator<` creates a `lambda_functor` of type:

```
lambda_functor<
    lambda_functor_base<
        relational_action<less_action>,
        tuple<placeholder<FIRST>, const int>
    >
>
```

2. The unary function call operator of this class is called with the variable `i` of type `int`. The return type is resolved to `bool` with the instantiation

```
lambda_functor::inherited::sig<tuple<int&> >::type
```

This return type and the arguments are forwarded to the `call` function of the base class:

```
lambda_functor_base<
    relational_action<less_action> >,
    tuple<lambda_functor<placeholder<FIRST>, const int>
>::call<bool>(i, const_null_type(), const_null_type());
```

3. The `call` function calls the `select` function to choose between the bound argument stored in the tuple member and the argument `i` supplied as a parameter:
- For the first argument `select(get<0>(args), a, b, c)` we end up making the call `select(_1, i, const_null_type(), const_null_type())` which returns a reference to `i`.
 - For the second argument, the constant `0` stored in the tuple is returned. The select call becomes: `select(0, i, const_null_type(), const_null_type())`.
4. The underlying target function, the less than operator, is called with the substituted argument list as `i < 0` and the result is returned all the way back to the caller of the `lambda_functor::operator()`, which concludes our example.

As we mentioned, the code examples presented are simplified from the actual library code. The following simplifications were made:

- The way bound arguments are stored in the tuples has to accommodate to some special cases in the C++ type system, e.g., that arrays and function types cannot be stored as non-reference types. Also, `ref` and `cref` (see Section 2.2) affect how bound arguments are stored. Consequently, instead of using the `make_tuple`, we construct the tuples with explicit constructor calls and map the tuple argument types via a traits class.
- There are additional layers in the return type deduction which we did not show. One layer copes with cases where evaluating a lambda functor results in another lambda functor. These cases may occur when a lambda functor is passed as a parameter to another lambda functor, or when the `protect` function, discussed in Section 3.5, is used. Another layer provides a point for the user to provide her own type rules for user defined operators (see 3.6).

5. About performance and use

A leading principle in C++ has been the “zero-overhead” rule, stating that writing code on a more abstract level should create no performance penalties [19]. A quote from Stroustrup defines zero-overhead informally [20]: “There is no way of writing equivalent C code that runs faster or generates smaller code.”

The LL tries to be obedient to this principle; the layers in the library consist of small inlined functions, which merely redirect their parameters delegating the actual work forward, ultimately all the way outside of the LL. Due to inlining, it is possible for the compiler to optimize away all overhead of using STL algorithms and lambda functors compared to hand written loops. The extent to which this is true in practice varies between compilers. In any case, no change for the worse is to be expected if lambda expressions are used instead of the more traditional STL tools (`plus`, `minus`, `bind1st`, `bind2nd`, `compose1`, etc.).

We performed a set of tests to assess this argument. The tests measure the *abstraction penalty* of lambda expressions and traditional unnamed STL function objects compared to hand-written function object classes with the same functionality. The term “abstraction penalty” was coined by Alex Stepanov, and it is calculated as the ratio of the running time of a code which takes advantage of certain abstractions, and the running time of a lower-level code which has equivalent functionality but does not use the abstractions.

The tests were run on a computer equipped with a 1.5 GHz Intel Pentium 4 processor with 256 MB of memory and 256 KB cache. We used the GCC 3.0.4 and KAI C++ 4.0f compilers. The optimization flag for the GCC compiler was `-O3` and for KAI C++ the flags `+K3 -O3 -inline_generated_space_time=20†` were used. In both tests we measured the execution times of the standard algorithm invocation:

```
std::transform(a.begin(), a.end(), b.begin(), F)
```

for different unary function objects `F`. In the first test, `a` and `b` were of type `std::vector<int>`, in the second test of type `std::vector<double>`. The size of the vectors was 100 elements, small enough to fit into the processor cache. The `transform` function was called repeatedly for each measurement.

In the first test, `F` was a function that multiplied the argument by itself repeatedly. The simplest case was the identity function, from which we went up to the case with 35 terms. Hence, the lambda expression cases for `F` were `_1`, `_1 * _1`, `_1 * _1 * _1`, etc. The corresponding hand-written function object classes were defined in the obvious way. We show the case with three terms:

```
struct expression_3 {
    double operator()(const int& x) { return x * x * x; }
};
```

The results of the first test are shown in Figure 5. The times are relative to the execution time of the hand-written identity function. The 3rd and 6th columns show the abstraction penalties of lambda expressions, which are negligible for both compilers.

In the second test, we used another set of expressions, which contained both bound and open arguments. We picked somewhat arbitrary arithmetic expressions with increasing complexity. The expressions are listed in Figure 6. The expressions were written as lambda expressions,

[†]KAI C++ is by default less eager to inline compiler generated functions, such as copy constructors, compared to functions explicitly declared inline, or to functions defined inside a class. This parameter makes KAI C++ to treat compiler generated functions as if the programmer had explicitly written out the function.

n	KAI C++			GCC		
	T_L	T_F	T_L/T_F	T_L	T_F	T_L/T_F
1	1.08	1.00	1.08	0.95	1.00	0.95
2	2.15	2.10	1.02	1.44	1.45	0.99
3	4.81	4.80	1.00	3.26	3.15	1.04
4	7.39	7.39	1.00	5.02	5.10	0.98
5	12.03	12.07	1.00	8.17	8.04	1.02
10	35.57	35.70	1.00	24.18	23.78	1.02
15	65.42	65.45	1.00	44.33	43.71	1.01
20	95.25	95.47	1.00	64.66	63.64	1.02
25	124.92	125.35	1.00	85.10	83.57	1.02
30	154.91	155.19	1.00	105.54	103.50	1.02
35	184.95	184.76	1.00	125.94	123.44	1.02

Figure 5. The measured running times of `std::transform(a.begin(), a.end(), b.begin(), F)`, where `F` is a unary function object performing n multiplications with its argument, `a` and `b` are of type `std::vector<int>`. T_L columns list times with `F` written as a lambda expression, T_F columns with `F` as a hand-written named function object class. The times are shown in arbitrary units relative to T_F in the $n = 1$ case (the identity function). The columns T_L/T_F show the abstraction penalty of lambda expressions.

as hand-written function object classes, and as traditional unnamed STL function objects. We do not show the last category, as the expressions get rather complex. For example, the expression 4 in Figure 6 contains 7 calls to `compose2`, 8 calls to `bind1st` and altogether 14 constructor invocations for creating `multiplies`, `minus` and `plus` objects. Note, that Figure 6 shows the execution times only for the hand-written function objects, again relative to the simplest case. For lambda expressions and traditional unnamed STL function objects we only show the abstraction penalties.

Our tests suggest that the LL does not introduce a loss of performance compared to traditional unnamed STL function objects. In most test cases there was no significant difference, and in few cases LL was clearly faster (see expressions 4 and 5 in Figure 6). Evaluating a lambda functor, or a traditional unnamed STL function object, consist of a sequence of calls to small functions that are declared inline. If the compiler fails to actually expand these functions inline, the performance can suffer. This seems to be the case in expression 4. Although the above tests do not show this happening for lambda expressions, we have experienced this for some seemingly simple expressions. In fact, this is the reason for using the `-inline_generated_space_time=20` parameter for the KAI C++ compiler.

To sum up the discussion about performance, with a reasonable optimizing compiler, one should expect the performance characteristics of lambda expressions to be comparable to traditional unnamed STL function objects. With simple expressions the performance can be expected to be close to that of explicitly written function objects.

i	KAI C++			GCC		
	T_F	T_L/T_F	T_C/T_F	T_F	T_L/T_F	T_C/T_F
1	1.00	1.06	1.09	1.00	1.13	1.25
2	1.43	1.01	1.00	1.45	1.12	1.19
3	2.13	1.03	1.03	2.04	1.03	1.12
4	4.95	0.99	4.27	5.01	1.14	1.13
5	3.43	0.90	0.90	3.56	1.20	1.73

i	expression
1	ax
2	$ax - (a + x)$
3	$(ax - (a + x))(bx - (b + x))$
4	$((ax - (a + x))(bx - (b + x))(ax - (b + x))(bx - (a + x))$
5	$x > 100 \ \&\& \ x < 1000$

Figure 6. Performance comparisons between unnamed functions written as lambda expressions, and using the traditional STL tools. The lower table shows a list of expressions, the upper table shows measured execution times for these expressions realized as hand-written named function object classes (T_F) and the abstraction penalties of lambda expressions (T_L/T_F) and traditional unnamed STL function objects (T_C/T_F). The number in the column i of the upper table refers to the expressions in the lower table.

Some design decisions were influenced by the optimization capabilities of compilers. Particularly, the `call` functions have a separate argument slot for each actual argument that the lambda functor is called with. An alternative would be to create a tuple object out of the arguments in the function call operator of `lambda_functor`. This would simplify the `call` functions, as they would only have one argument, instead of three, or whatever the upper limit for the number of arguments is. The KAI C++ compiler is capable of optimizing these intermediate objects away, but not other compilers we experimented with. The relative cost of constructing the argument tuple is highest for the very simple expressions—which are just the functions one would most likely want to write as lambda expressions. If a tuple was used instead of a separate argument slots we measured up to four times slower runtimes for expressions like:

```
vector<int> a;
...
for_each(a.begin(), a.end(), _1 * _1)
```

Another issue is the impact the LL has on compile times. Expression templates usually involve recursive template instantiations and can slow down compilation considerably. The LL is no exception, especially deeply nested lambda expressions can be slow to compile.

As another downside, compilation error messages that result from invalid lambda expressions can be very hard to comprehend. Even a relatively simple lambda functor can have a type

that spans several lines in an error message. This is a general concern with heavily templated C++ code [21].

5.1. Relation to other work

The LL originated from combining, extending and generalizing the functionalities of the *Expression Template library* (ET) by Powell and Higley [10] and the *Binder Library* by Järvi [6]. The ET library overloaded a smaller set of operators and required the placeholders to be typed, leading to function objects with fixed argument types. Further, there was no return type deduction: ET library assumed that the return type was the same as the first argument type. The Binder Library contained the bind expression part of the LL in a slightly more restricted form; the first argument for bind expressions was not allowed to be an arbitrary bind expression.

Other related work includes the FACT [22] and FC++ [11, 12] libraries, both developed coevally with the LL. The basic idea behind the FACT lambda functions is quite similar to the LL counterparts, although the syntax is different. Compared to LL, FACT supports a smaller set of operators, it has no support for control structures, constructors etc. FACT deliberately allows no side effects in lambda functions, which means that for example the assignment operators are not supported. FACT lambda functions are ‘expression template aware’ (see [23]), while basic LL lambda functions are not. Further, FACT provides other features in addition to lambda functions, such as lazy lists.

The FC++ is another library adding functional features to C++; it more or less embeds a functional sublanguage to C++. The FC++ is not only a library, but also a framework for writing higher-order polymorphic functions in C++. One key innovation in FC++ is its type system, that relies on the existence of signature classes (`sig` templates) within each function object class. This finding has influenced the return type deduction mechanism of the bind expressions in LL (see Section 2.5.3).[‡] Another notable feature of FC++ is the possibility to define variables for storing polymorphic function objects, which is not directly feasible due to the complex type of such functions. This is convenient, even though the feature comes with some cost: the function becomes dynamically bound, and the return type and the argument types of the function must be defined explicitly by the client. The *Function Library* in *C++ Boost* [24] has later packaged this feature in a form that can be combined with other libraries, e.g., with the LL. The FC++ and LL are complementary efforts. The FC++ focuses on the back-end part of functional programming in C++, revealing the implementation to the programmer, and requiring the programmer to write the function objects in a specific manner to be usable within the FC++ framework. On the other hand, the LL focuses on the front-end, providing the syntax for defining unnamed functions.

Other work on combining functional programming with C++ include the work by Läufer [25], which predated the FC++ library, and provides a limited subset of the functionality of FC++.

[‡]In earlier versions of the LL, the return type deduction was equally powerful but less convenient for this purpose, as adding a new function object class to the framework required a specialization of another return type template outside the function object class.

Building lambda abstractions as preprocessor macros was proposed in [26]. A chapter on generalized functors in [27] suggests some improvements over the classic STL function objects.

The LL implementation has been influenced by work of others. The return type promotion code of arithmetic types is modeled after that used in the Blitz++ library [28]. Unlike the early versions of the LL, many of the type functions now rely on the Boost *type_traits* library [29].

More recent work [13] related to LL has shown that LL style lambda functors can have true local variables. With a construct like:

```
locals<T1, T2, ..., TN>(t1, t2, ..., tn)(lambda_expression)
```

one defines local variables of types T1, ..., TN and initializes them with values t1, ..., tn. Analogous to the placeholders, the local variables have predefined names, such as `loc1`, `loc2`, etc. which can be used to refer to the variables in the enclosed lambda expression.

There has been some discussion about adding a `typeof` operator (for querying a type of an expression statically) into the standard C++. This is an open issue, but there is some support for this initiative in the C++ community. The Lambda Library would benefit from a full compiler supported `typeof` expression as it would eliminate a great deal of the complex return type deduction code.

To summarize LL's relation to other related libraries, to our understanding, libraries like FACT and FC++ take the functional features in a 'pure' form. For example, the FC++ advises against using function objects with side effects as part of the framework, and FACT does not provide the `++` and `--` operators as they have side effects. The LL implements lambda functions adjusted for a better fit to C++; particularly, if the underlying target functions have side effects, the LL makes no effort in trying to prevent them. On the contrary, the library tries to make the lambda functions as transparent as possible and avoids changing the semantics of the target functions. Our foremost goal with this library is to provide lambda functions which match perfectly with the STL style of programming.

Original innovations of the LL (or its predecessors) include the untyped placeholders giving simultaneously polymorphic lambda functors, partial function application within arbitrary argument positions, and arbitrary function composition of different types of lambda functors. The LL also shows that the return type rules of the C++ operators can be captured in a set of type deduction templates to the extent that is satisfactory for practical programming.

6. Conclusion

With the Lambda Library we hope to provide a valuable set of tools for working with STL algorithms. The LL removes several restrictions and simplifies the use of the STL in many ways. The users of the LL have a natural way to write simple functors cleanly. Teachers of STL algorithms can have students writing clear code quickly: it is easier to explain how to use the LL than the current alternative of `ptr_fun`, `mem_fun`, `bind1st`, etc. and it extends better to the more complex problems (cf. `bind3rdAnd4th`). Further, the LL introduces a set of entirely new possibilities for STL algorithm reuse: The LL makes it possible to loop through multiple nested containers. Exceptions can be thrown, caught and handled within the functor, and the looping in the STL algorithm can be continued. All the above features are built with

standard C++ templates and do not change the design model of the language or require an additional preprocessing step.

We are aware of the downsides of the library: complexity, increased compile times and difficult error messages. These are problems with classic STL as well, albeit slightly more pronounced in the LL. Despite its problems, STL became extremely popular. The extensions that the LL adds to STL have potential for being adopted in wide use as well. There is always room for improvement, but we believe that in terms of generality, ease of use and intuitiveness of writing function objects for STL algorithms, the tools in LL are getting close to what can be achieved without changes to the core language.

The Lambda Library is part of the *C++ Boost* library collection and is freely downloadable at <http://www.boost.org>.

7. Acknowledgments

No project exists in a vacuum. We are indebted to those who have gone before us and helped us along the way with code, comments and suggestions. This list of people in no particular order includes Jeremy Siek, William Kempf, Peter Higley, Valentin Bonnard, Dave Abrahams, Joel de Guzman, Peter Dimov, Douglas Gregor, Mat Marcus, Martin Weiser and Aleksey Gurtovoy.

REFERENCES

1. A. A. Stepanov and M. Lee. The standard template library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, April 1994. <http://www.hpl.hp.com/techreports>.
2. M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, Reading, MA, USA, 1998.
3. International Standard, Programming Languages – C++, ISO/IEC:14882, 1998.
4. Interactive Software Engineering Inc. *Agents, iterators and introspection*, 2000. <http://www.eiffel.com> (Information, Technical papers).
5. The SGI Standard Template Library. <http://www.sgi.com/tech/stl>, 2002.
6. J. Järvi. C++ function object binders made easy. In *Proceedings of the Generative and Component-Based Software Engineering'99*, volume 1799 of *Lecture Notes in Computer Science*, pages 165–177, Berlin, Germany, August 2000. Springer.
7. V. Simonis. Adapters and binders - overcoming problems in the design and implementation of the C++ STL. *ACM SIGPLAN Notices*, 35(2):46–53, February 2000.
8. T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.
9. J. Järvi and G. Powell. Side effects and partial function application in C++. In *Proceedings of the Multiparadigm Programming with OO Languages Workshop (MPOOL'01) at ECOOP 2001*, John von Neumann Institute of Computing series, pages 43–60, 2001.
10. G. Powell and P. Higley. Expression templates as a replacement for simple functors. *C++ Report*, 12(5), May 2000.
11. B. McNamara and Y. Smaragdakis. Functional Programming in C++. In *Proceedings of The 2000 International Conference on Functional Programming (ICFP)*, New York, NY, USA, 2000. ACM Press. <http://www.cc.gatech.edu/~yannis/fc++>.
12. B. McNamara and Y. Smaragdakis. Functional Programming in C++ using the FC++ library. *ACM SIGPLAN Notices*, 36(4):25–30, April 2001.
13. J. de Guzman. Phoenix Library, part of Spirit Parser Library, 2002. <http://spirit.sourceforge.net/>.
14. J. Järvi and G. Powell. The Boost Lambda Library. <http://www.boost.org>, 2002.
15. K. Czarnecki and U. Eisenecker. *Generative Programming : Methods, Tools, and Applications*. Addison-Wesley, 2000.

-
16. S. W. Haney, J. Crotinger, S. Karmesin, and S. Smith. Pete, the portable expression template engine. *Dr. Dobbs's Journal*, pages 88–95, October 1999. <http://www.acl.lanl.gov/pete>.
 17. J. Järvi. Tuple types and multiple return values. *C/C++ Users Journal*, 19:24–35, August 2001.
 18. J. Järvi. Boost tuple library. <http://www.boost.org>, 2001.
 19. A. Koenig and B. Stroustrup. Foundations for native C++ styles. *Software – Practice and Experience*, 25(S4):S4/45–S4/86, December 1995.
 20. B. Stroustrup. Standard C++ for embedded systems programming. Keynote talk at Embedded Systems Conference, Chicago, March 1999. www.research.att.com/~bs/esc99.html.
 21. A. Alexandrescu. Better template error messages. *C/C++ Users Journal*, 17(3), March 1999. <http://www.cuj.com>.
 22. J. Striegnitz. The FACT! library home page, 2000. <http://www.fz-juelich.de/zam/FACT>.
 23. J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 2000. <http://oonumerics.org/tmpw00/>.
 24. D. Gregor. Boost function library. <http://www.boost.org>, 2001.
 25. K. Läufer. A framework for higher-order functions in C++. In *Proceedings of the Conference on Object-Oriented Technologies (COOTS)*, Monterey, CA, June 1995. USENIX. <ftp://ftp.math.luc.edu/pub/lauffer/papers/functoids.ps.gz>.
 26. O. Kiselyov. Functional style in C++: Closures, late binding, and lambda abstractions. poster presentation at International Conference on Functional Programming, Baltimore MD, <http://okmij.org/ftp/c++-digest/Functional-Cpp.html>, September 1998.
 27. A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.
 28. T. Veldhuizen. The Blitz++ library home page, 1999. <http://www.oonumerics.org/blitz>.
 29. J. Maddock. Boost type traits. <http://www.boost.org>, 2002.