The Lambda Library

Lambda Abstraction in C++

Jaakko Järvi

jaakko.jarvi@cs.utu.fi

Gary Powell

Sierra On-Line Ltd.
gary.powell@sierra.com

Turku Centre for Computer Science
TUCS Technical Report No 378
November 2000

ISBN 952-12-0758-2
ISSN 1239-1891

Abstract

The Lambda Library (LL) adds a form of *lambda abstraction* to C++. 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 STL algorithms. The LL offers significant improvements, in terms of generality and ease of use, compared to the binders and functors in the C++ standard library.

Keywords: lambda abstraction, generic programming, C++ templates

The Lambda Library

Lambda Abstraction in C++

Jaakko Järvi

jaakko.jarvi@cs.utu.fi

Gary Powell

Sierra On-Line Ltd.
gary.powell@sierra.com

Turku Centre for Computer Science
TUCS Technical Report No 378
November 2000

ISBN 952-12-0758-2
ISSN 1239-1891

1 Introduction

The ability to define *lambda functions*, i.e. unnamed functions, is a standard feature in functional programming languages. For some reason, this feature does not appear in any of the mainstream procedural or object-oriented languages. This article introduces the *Lambda Library* which fixes this 'omission' for C++.

The Lambda Library (LL) is a C++ template library implementing a form of lambda abstraction for C++. The library is designed to work with the *Standard Template Library* (STL) [SL94], now part of the C++ Standard Library [C++98]. The paper assumes some familiarity with the STL.

The article consists of two parts. In the first part we explain how to use the library, in the second part we go over the implementation details in all their glory.

1.1 Motivation

Typically STL algorithms operate on container elements via functions and *function objects* passed as arguments to the algorithms. An object of a class with a function call operator is a function object. Function objects are sometimes called *functors* as well.

The STL contains predefined function objects for some common cases (such as `plus`, `less` and `negate`). In addition, it contains *adaptors* for creating function objects from function pointers etc. Further, it contains two *binder* templates `bind1st` and `bind2nd`. With binders, you can create a unary function object from a binary function object by *binding* one of the arguments to a constant value. Some STL implementations contain function composition operations as extensions to the standard [STL00].

The binder templates can be seen as kind of *currying* operators, but together with negaters and function composition operators the goal of all these tools is clear: to make it possible to specify unnamed functions in a call to an STL algorithm. Still in other words, the goal is to be able to pass code fragments as arguments to functions. However, the tools in the Standard Library provide only a small step towards this goal. Unnamed functors built as compositions of standard function objects, binders, adaptors etc. are very hard to read in all but the simplest cases. Moreover, the 'lambda abstraction' 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. See [Jär00] for a more in-depth discussion about these restrictions.

The Lambda Library solves these problems. The syntax is intuitive and there are no (well, fewer) arbitrary restrictions. 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 are 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'll take an example to demonstrate what LL's impact can be on code using STL algorithms. The following example is an extract from the documentation of one STL implementation:

... calculates the negative of the sines of the elements in a vector, where the elements are angles measured in degrees. Since the C library function sin takes its arguments in radians, this operation is the composition of three operations: negation, sin, and the conversion of degrees to radians.

```
vector<double> angles;
vector<double> sines;
const double pi = 3.14159265358979323846;
  ...
assert(sines.size() >= angles.size());
transform(angles.begin(), angles.end(), sines.begin(),
            compose1(negate<double>(),
              compose1(ptr_fun(sin),
                bind2nd(multiplies<double>(), pi / 180.0))));
```

Using LL constructs, the `transform` function call can be written as:

```
transform(angles.begin(), angles.end(), sines.begin(),
            - bind(sin, free1 * pi / 180.0));
```

The `operator-` replaces the call to `negate`, `bind` replaces `compose1`, `ptr_fun` becomes unnecessary, along with `bind2nd`, and the call to `multiplies` is replaced with the `operator*`. The argument `free1` is a placeholder representing the parameter of the function object. At each iteration, it will be substituted for the element from the container of angles. Notice how the code using the Lambda Library is much easier to read.

1.2 About expression templates

The LL implementation is based on *expression templates*. 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. In the LL case, this manipulation means passing the expression object as a parameter to a function, and substituting the actual arguments for the placeholder objects.

There are scarcely any introductory articles on expression templates. The original article by Veldhuizen [Vel95] described expression templates tailored for vector and matrix expressions. Such expression templates are discussed also in [CE00, Chapter 10] as well as in [HCKS99]. The article on the *Expression Template Library* (ET) [PH00] explains expression templates which are closer to the LL implementation. We will spend more time on LL expression templates in the second part of this article.

Expression templates are a complex topic, so don't be discouraged if you don't fully understand them. Just like the I/O streams library you can use them anyway. In part I we are going to try and show you what you can do with them, not the internals of how they work.

1.3 Terminology

The expression `- bind(sin, free1 * pi / 180.0)` shown in section 1.1 is an expression template. This is according to commonly established C++ terminology. However, to emphasize the analogy to functional programming we call this a *lambda expression*, as it in effect defines an unnamed function, or more precisely, an unnamed function object. We use the term *lambda functor* to specifically refer to the function object resulting from a lambda expression. To relate our terminology with existing concepts, as we understand them, the term expression template covers the whole machinery for creating expression objects and calling them. We need a finer segregation between concepts; hence we make a distinction between lambda expressions and lambda functors.

Part I
Using the Lambda Library

2 Basic usage

This section describes the basic rules for writing lambda expressions consisting of operator invocations and different types of functions and function-like constructs. Note that there are exceptions to these basic rules. Some operators in C++ have special semantics, which is reflected in the semantics of the corresponding lambda expressions as well. We cover these special cases in section 3.

2.1 Introductory examples

It is easiest to understand how to use the LL by looking at some examples. So let's start with some simple expressions and work up. First, we'll initialize the elements of a container (e.g. a `list`) to the value `1`:

```
list<int> v(10);
for_each(v.begin(), v.end(), free1 = 1);
```

In this example `free1 = 1` creates a lambda function which assigns the value `1` to every element in `v`; remember that `free1` is a placeholder, an empty slot, which will be filled with a value at each iteration. Next, we create a container of pointers and make them point to the elements in the first container `v`:

```
list<int*> vp(10); // a container of pointers
transform(v.begin(), v.end(), vp.begin(), &free1);
```

Here we take the address of each element in `v` (with `&free1`) and assign it to the corresponding element in `vp`. Now lets change the values in `v`. For each element we call some function `foo` passing the original value of the element as an argument to `foo`:

```
int foo(int);
for_each(v.begin(), v.end(), free1 = bind(foo, free1));
```

Next we'll sort the elements of `vp`:

```
sort(vp.begin(), vp.end(), *free1 > *free2);
```

In this call to `sort`, we are sorting the elements by their contents in descending order. Note that the lambda expression `*free1 > *free2` contains two different placeholders, `free1` and `free2`. Consequently, it creates a binary lambda functor. When this functor is called, the first argument is substituted for `free1` and the second argument for `free2`. Finally we'll output the sorted content of `vp` separated by line breaks:

```
for_each(vp.begin(), vp.end(), cout << *free1 << endl);
```

After seeing these examples it should be obvious how to write the lambda expression to, say, increment each element in `v` by 6 times the result of the call to a zero-argument function named `rand`:

```
for_each(v.begin(), v.end(), free1 += 6 * bind(rand));
```

See this is easy. We must admit that there are some traps and we'll try and point them out as we go along. But in general you will have no trouble using this new library.

2.2 Placeholders

In lambda functions occurring in *lambda calculus* and in functional programming languages the formal parameters are commonly named within the lambda expression with some explicit syntax.For example, the following definition gives the names x and y to the formal parameters of an addition function:

$\lambda xy.x + y$

The C++ counterpart of the above expression is written as:

```
free1 + free2
```

In the C++ version the formal parameters, i.e. the placeholder variables, have predefined names. There is no explicit syntactic construct for C++ lambda expressions; the use of a placeholder variable in an expression implicitly turns the expression into a lambda expression.[1]

So far we have used the placeholders `free1` and `free2`. The LL supports one more: `free3`. This means that lambda functors can take one, two or three arguments passed in by the STL algorithm; zero parameters is possible too. No STL algorithm accepts a functor with the number of arguments greater than two, so the three placeholders should be enough. In fact, the third placeholder is a necessity in order to implement all the features of the current library (see section 3.8, which introduces `freeE`, another incarnation of `free3`).

The placeholders may seem somehow magical, but they are just normal variables. The variables themselves are not important but rather their types are. The types serve as tags, which allow us to spot the placeholders from the arguments stored in a lambda functor, and later replace them with the actual arguments that are passed in the call to the lambda functor. The LL provides typedefs for the placeholder types, so it is easy to define the placeholder names to your own liking. For example, to use `arg1`, `arg2` and `arg3` instead of `free1`, `free2` and `free3`, you can write:

```
free1_type arg1;
free2_type arg2;
free3_type arg3;
```

Veldhuizen [Vel95] was the first to introduce the concept of using placeholders in expression templates. The LL placeholders are somewhat different. You may have noticed from the previous examples that we do not specify any types for the arguments that the placeholders stand for. A placeholder leaves the argument totally open, including the type. This means that the lambda

[1] This doesn't hold for all expressions, which we will explain in the sequel.

functor can be called with arguments of any type for which the underlying function makes sense.

Consider again the first example we showed in section 2:

```
for_each(v.begin(), v.end(), free1 = 1);
```

The lambda expression `free1 = 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`.

Since the type of the placeholder argument is left open, it is obvious that the return type of the lambda functor is not known either. Not to worry, the LL has a type deduction system that figures out the return type when the lambda functor is called. It covers operators of built-in types and 'well-behaved' operators of user defined types; and for your user defined operators with unorthodox return types, the deduction system is easy to extend. We will cover this aspect of the LL in the latter part of the paper.

2.3 Functions as lambda expressions

Obviously, the use of a placeholder in a function call does not turn the invocation into a lambda expression. As the examples above show, 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. Where a placeholder is used in place of an actual argument, we say that the argument is *unbound* or *free*.

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. Note that we use the term *target function* with other types of lambda expressions as well, always to denote the underlying operation of the lambda expression.

2.3.1 Function pointers as targets

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

```
X foo(A, B, C); A a; B b; C c;
  ...
bind(foo, free1, free2, c)
bind(&foo, free1, free2, c)
bind(foo, free1, free1, free1)
bind(free1, 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 demonstrates that a certain placeholder can be used multiple times in a lambda expression. The argument will be duplicated in each place that the placeholder is used. For this bind expression to make sense, and to compile, the argument to the resulting unary lambda functor must be implicitly convertible to `A`, `B` and `C`. The fourth bind expression shows that even the target function can be left unbound; 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, free1)(i); // error
  ...
void (*pf1)(int) = &foo;
bind(pf1, free1)(i); // ok
```

Also, there is no way to gather the default arguments to a function:

```
void goo(int, int = 10);
bind(goo, free1)(i); // error: goo takes two arguments
```

2.3.2 Function objects as targets

Function objects can also be used as target functions. The return type deduction system requires that the function object class defines the return type of the function call operator as the typedef `result_type`. This is the convention used with the function objects in the Standard Library. For example:

```
struct A {
 typedef B result_type;
 B operator()(X, Y, Z);
};
```

The above function object can be used like this:

```
A a; X x; Y y; Z z; list<A> la; list<Z> lz;
  ...
for_each(lz.begin(), lz.end(), bind(a, x, y, free1));
for_each(la.begin(), la.end(), bind(free1, x, y, z));
```

The function call operator can be overloaded within a class. However, the return type deduction system can handle only one return type per function object class. Consequently, all the overloaded function call operators within one class must have the same return type, if you want to be able to use them as target functions (there is a way around this, see section 6.7.1).

2.3.3 Member functions as targets

The form of the bind expression with member function targets is slightly different. By convention, we have chosen to declare the `bind` functions with the following format:

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

If the first argument is a pointer to a member function of some class `A`, the second argument is the *target object*, that is, an object of type `A` for which the member function is to be called. Actually, 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); A a; A* ap; vector<int> ints;
  ...
// reference is ok:
find_if(ints.begin(), ints.end(), bind(&A::foo, a, free1));

// pointer is ok:
find_if(ints.begin(), ints.end(), bind(&A::foo, ap, free1));
```

The functionality is identical in both cases. Similarly, if the object argument is free, the resulting functor can be called both via a pointer or a reference:

```
list<A> refs; list<A*> ptrs;
find_if(refs.befin(), refs.end(), bind(&A::foo, free1, 1));
find_if(ptrs.begin(), ptrs.end(), bind(&A::foo, free1, 1));
```

Similar to other types of bind expressions, the `target-member-function` can be left open as well.

2.4 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 special cases and restrictions:

- The return types cannot be chosen freely while overloading operators `->`, `new`, `delete`, `new[]` and `delete[]`. Consequently, we can't overload them directly for lambda expressions.
- It is not possible to overload the `.`, `.*`, and `?:` operators in C++.
- The assignment and subscript operators must be defined as member functions, which creates some asymmetry to lambda expressions. For example:

  ```
  int i;
  free1 = 1; // a valid lambda expression
  i = free1; // error, no assignment from free1_type to int
  ```

  A workaround for this situation is explained in section 2.5.
- As stated in section 2.2, the return type deduction system may not handle all user-defined operators. In such cases the deduction system can either be extended, or temporarily overridden by explicit type information (see section 6.7).

2.5 Delayed constants and variables

Lambda functors are built with function calling rules: arguments that can be evaluated before the lambda functor creation function is called, will be. This may not be what we want. We commonly need to prevent the evaluation of a variable, or sometimes even a constant, and create a lambda functor out of the variable, or constant, instead. We call such lambda functors *delayed variables*, or *delayed constants*, respectively. For example, the following line outputs a space followed by the elements of the container `a`:

```
for_each(a.begin(), a.end(), cout << " " << free1);
```

To output a space separated list of elements, we need to turn the constant `" "` into a lambda expression with the `constant` function:

```
for_each(a.begin(), a.end(), cout << constant(" ") << free1);
```

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(a.begin(), a.end(), var(count)++);
```

The variable `count` is delayed; it 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(a.begin(), a.end(), vcount++);
```

This feature is useful if the same variable appears repeatedly in a lambda expression. This is relatively common with some of the *control lambda expressions*, see section 3.5. In general, delayed variables are in frequent use within control lambda expressions.

Above, in section 2.4 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 = free1; // error, no assignment from free1_type to int
var(i) = free1; // ok
```

2.6 Controlling how bound arguments are stored

We have stated that bound arguments are stored in the lambda functor. However, there are a few alternative ways of doing it. Basically the lambda functors can store temporary copies of the arguments, or hold const or non-const references to them. The default is temporary copies. 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; (free1 + i)(i = 10);
```

As said, this is the default, and for some expressions it makes more sense to store the arguments as references. As an example, consider the lambda expression `i += free1`. 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 left argument of the combined assignment operators (`+=`, `*=`, etc.) are stored as references to non-const. Further, to make the streaming operators (`<<` and `>>`) intuitive, the stream argument is stored as a reference. Also, as array types cannot be copied, lambda functors store references to arguments that are of array types.

In lambda functors created with bind expressions, the default is again to store temporary copies. This is a reasonable default, as it prevents side-effects via bound arguments. However, we may have target functions where the side-effects are deliberate. Further, we may have argument types that cannot be copied, or are very expensive to copy. For these reasons the LL provides two simple functions for overriding the default storing mechanism: wrapping a bound argument with the `ref` function states that the lambda functor should store a non-const reference to the argument, and respectively, the `cref` function states that a const reference should be used. For example:

```
A a; int i; // suppose A is a class that can't be copied
void foo(const A& a);
void rotate(int& j) { j < 10 ? ++j : j = 0; }
  ...
bind(foo, cref(a)) // cannot copy a, store a const reference instead
bind(rotate, ref(i)) // store a reference to i
```

The implementation of `ref` and `cref` is not discussed in part II, but rather we describe the basic idea here briefly (for more details, see [Jär00]). These functions create wrapper objects that hold a reference to the wrapped argument. The functions and operators that create lambda functors recognize such wrapper types and select the storing method appropriately. Further, the wrapper object has a conversion operator to the wrapped reference type, hence the object can be used transparently in place of the wrapped argument. Finally, note that `ref` and `cref` do not create lambda functors, and are thus different from `var` and `constant` explained in section 2.5.

3 Advanced techniques (templates gone wild)

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 use some of the C++ specific operators in lambda expressions, how to write control structures as lambda expressions and how to construct and destruct objects in lambda expressions; we even show how to do exception handling in lambda expressions. We believe that in addition to the 'typeless' placeholders and the unique return type deduction system, these extra features really set the LL apart from its predecessors.

3.1 Comma operator

The LL overloads the comma operator for sequencing lambda expressions together, as did the ET library [PH00]. The character ";" is reserved by the C++ language to mean 'end of statement'. For this library we would like to have it mean, 'end of this lambda expression'. Unfortunately it just isn't possible, so we are left with `operator,`.

Since comma is also the separator between function arguments, extra parenthesis are sometimes necessary to write syntactically correct lambda expressions:

```
for_each(a.begin(), a.end(),
           (cout << free1 << endl, clog << free1 << endl));
```

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 `a` is guaranteed to be first written to `cout` and then to `clog`.

Note that the short circuiting rules for the operators `&&`, and `||` are respected as well. For example, the following code sets all negative elements of some container `a` to zero:

```
for_each(a.begin(), a.end(), free1 < 0 && free1 = 0);
```

See [Mey96, pp. 35–38] for a full discussion about the pitfalls when overloading comma and the short circuited logical operators.

3.2 Pointer to member operator

The `operator->*` is another special case. It can refer to either a member object, or a member function. In the case of a member function, this overloaded operator has duplicate functionality to using `bind`. Nevertheless, it may make the code easier to read. For example:

```
struct data {
  int x;
  int foo(int y);
};
vector<data*> v;

// count all elements, where the member x equals to zero:
count_if(v.begin(), v.end(), free1->*(&data::x) == 0);
```

```
// stream out the results of calls to data::foo(5):
for_each(v.begin(), v.end(),
           cout << (free1->*(&data::foo))(5));

// the same example using bind:
for_each(v.begin(), v.end(),
           cout << bind(&data::foo, free1, 5));
```

3.3 Function call operator

We have also overloaded `operator()` for the placeholders. We provide the zero through nine argument cases of `operator()(arg1, ..., arg9)`. The same functionality can be achieved using `bind`. The `operator()` has been overloaded because it may make your code easier to read and understand, which is what the LL is all about. Here is an example using `operator()`:

```
struct data {
  // typedef required for return type deduction
  typedef int result_type;
  int operator()(int y);
};
int y;
vector<data> v;
  ...
count_if(v.begin(), v.end(), free1(y) == 0);
```

We have not overloaded `operator()` for arbitrary lambda functors, since this would conflict with the `operator()` already defined in the lambda functor.

3.4 Conditional expression

Since it is not possible to overload the `?:` operator, we have created a function called `if_then_else_return`, which duplicates its functionality. For example:[2]

```
// if bit 0x001 is set, set bit 0x100, else set bit 0x010
for_each(a.begin(), a.end(),
  free1 |=
    if_then_else_return(
      (free1 & 0x001) == 0x001, 0x100, 0x010));
```

The result type rules for the `?:` operator are complex but we believe we have them encoded to the extent that it is possible.

[2]Since we have overloaded `operator|=`, the `for_each` loop is as efficient as a hand coded for loop. Unlike using transform which only uses `operator=`, the LL can use the `+=`, `-=` etc. operators which take advantage of not having to create a new object on the stack and then call the assignment operator.

3.5 Control lambda expressions

The idea of lambda expressions for control structures originates from the ET Library [PH00]. The LL implements the control lambda expressions of the ET library and adds more. In addition to `if_then`, `if_then_else`, `for_loop`, `while_loop`, and `do_while_loop`, the LL also has `switch` and `do_once`.

Control lambda expressions are function templates that create lambda functors that implement the behavior of some control structure. 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(free1 % 2 == 0, cout << free1));
```

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

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

Again, the arguments to the `for_loop` function are lambda functors. Let's take 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),
                  free1[var(i)] += 6));
```

Note the use of delayed variables to turn the `for_loop` arguments into lambda expressions.

As stated in section 2.5, 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, free1[vi] += 6));
```

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

3.6 Switch and do_once

The lambda expressions for `switch` control structures, as well as for `do_once`, 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* is a lambda expression creating a lambda functor with an integral return type. The different cases are created with `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. The `break` statements are implicitly part of each case: for example, `case_statement<1>(lambda_functor<A>& a)` generates the code:

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

We have specialized the `switch_statement` function for up to 9 case statements.

In case you do not recognize the `do_once` construct, it is a specialization of the code:

```
do {
  code; if (condition) break;
  code; if (condition) break;
  ...
  code;
} while(false);
```

The lambda expression equivalent to this is written as:

```
do_once (
  do_statement(lambda expression, break_if(condition)),
  do_statement(lambda expression, break_if(condition)),
  ... ,
  lambda expression
)
```

The `do_once` construct is similar in structure to the switch lambda expression: `do_statement` functions group the code into blocks, just as the `case_statement` functions in the switch structure. The last argument is the end of the do once structure, and thus a plain lambda functor; there is no need for another test/break block.

3.7 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. As in the case of the conditional operator, we have defined function templates to circumvent this restriction. The function template `new_ptr` creates a lambda functor that wraps a `new` invocation, `delete_ptr` respectively a lambda functor that wraps a deletion. For example:

```
int* a[10];
for_each(a, a+10, free1 = new_ptr<int>());
for_each(a, a+10, free1 = delete_ptr(free1));
```

The `new_ptr` function takes the type of the object to be constructed as an explicitly specified 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. The lambda functor created with `delete_ptr` first evaluates its argument (which is a lambda functor as well) and then calls `delete` on the result of this evaluation. We have also defined `new_array` and `delete_array` for `new[]` and `delete[]`.

To be able to write constructors as lambda expressions, we have to resort to a set of function templates again. We cannot use `bind`, since it is not possible to take the address of a constructor. Instead, we have defined a set of `constructor` functions which create lambda functors for constructing objects. The lambda expression

`constructor<`*type*`>(`*args*`)`

creates a lambda function which wraps the constructor call *type*`(`*args*`)` and returns the resulting object. The complementary function `destructor` exists as well. The following example reads integers from two containers, constructs pairs out of them and inserts them into a third container:

```
vector<pair<int, int> > v;
transform(x.begin(), x.end(), y.begin(), back_inserter(v),
          construct<pair<int, int> >(free1, free2));
```

3.8 Exception handling in lambda expressions

The LL allows you to create lambda functors that throw and catch exceptions. At first glance, this may seem a straightforward addition to the library

— after all, if you can implement switch statements with an arbitrary number of case blocks, why not try and catch blocks? Trust us, it is not that easy. We will explain more in the second part of the article (see section 9). Nevertheless, the usage is fairly easy. 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 resulting lambda functors catch the exceptions as references. The *lambda expression* within the `catch_exception` defines the actions to take if the exception is caught.

The last catch block can be either a call to `catch_exception<`*type*`>`, or to `catch_all`. Since it is not possible to write `catch_exception<...>`, we have used the function `catch_all` to mean `catch(...)`.

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.

The figure 1. demonstrates the use of the LL exception handling tools. The first catch block is for handling exceptions of type `foo_exception`. Note the use of the `free1` 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 `cout` and constructs and throws another type of exception (`bar_exception`). The `std::exception` carries a string explaining the cause of the exception. The explanation can be queried with the zero-argument `what` member function; `bind(&std::exception::what, freeE)` creates a lambda functor for calling the `what` function. Note the use of `freeE` as the argument. It is a special placeholder, which refers to the caught exception object within the handler body. `freeE` is not a fullfledged placeholder, but rather a special case of `free3`. As a consequence, `freeE` cannot be used outside of an exception handler lambda expression, and `free3` cannot be used inside of an exception handler lambda expression.[3] As `free1` and `free2` can be used inside a handler, in addition to `freeE`, we do not see

[3]Fair enough.

```
for_each(
  a.begin(), a.end(),
  try_catch(
    bind(foo, free1), //foo may throw
    catch_exception<foo_exception>(
      cout << constant("Caught foo_exception; foo argument = ")
           << free1
    ),
    catch_exception<std::exception>(
      cout << constant("Caught std::exception: ")
           << bind(&std::exception::what, freeE),
      throw_exception(constructor<bar_exception>(free1))
    ),
    catch_all(
      cout << constant("Unrecognized exception"),
      rethrow()
    )
  )
);
```

Figure 1: Throwing and handling exceptions.

the immediate threat of this being a true restriction in practice. In any case, illegal use of placeholders is caught by the compiler.

The last handler (`catch_all`) demonstrates rethrowing exceptions.

3.8.1 Why do exception handling at all?

Lambda expressions for exception handling were one of the last things we added to the Lambda Library, because we thought it was not that important, and because it was not that easy. Looking back, it was worth the effort.

Exception handling made the promise of giving you another chance; fix things up and maybe try again. But the standard functors with the standard algorithms don't give that chance. If the functor passed to an algorithm throws, you can't restart the algorithm where it was when the exception was thrown. In other words, if there is a possibility that the functor throws, you can't make exception safe calls to STL algorithms — unless of course you rewrite the functor to handle the exceptions. The LL gives you an easy way to write exception safe code. Placing a try/catch block inside the algorithm allows you to decide whether to continue or bail.

3.9 Nesting STL algorithms with lambda expressions

At the time of writing this, we are extending the library to allow nesting of STL algorithms. There are some issues which makes this complicated, and we haven't affirmatively decided on all the details yet. Consequently, this section describes work in progress and the exact syntax for nesting STL algorithms may change in the future versions of the library.

In section 3.5 we showed an example using `for_loop` as the function object passed to the `for_each` algorithm. We'll 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),
                  free1[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 thus cannot be passed as a parameter. To circumvent this, we have copied the interface of all the STL algorithms into a subnamespace `LL`, where the names of the standard algorithms refer to our own lambda functors, which implement the functionality of the corresponding algorithms (by calling the standard function template). Using a nested `for_each`, the above example becomes:

```
for_each(a, a+5, LL::for_each(free1, free1 + 10, free1 += 6));
```

Notice the reuse of `free1`. In the first two arguments of the inner `for_each` it refers to the elements over which the outer `for_each` iterates. In the third argument, the same placeholder refers to the elements in the inner `for_each`. What is going on? Well, `free1` is just a placeholder so we can reuse it wherever a placeholder makes sense. The first two arguments to `LL::for_each` are just plain old lambda functors, which the `LL::for_each` functor evaluates at each iteration and passes the results forward to `std::for_each`. The third argument is a lambda functor as well, but it is not evaluated, but rather passed on to `std::for_each` as such.

4 Getting, installing and using the library

To get the library, visit the LL web-site `http://lambda.cs.utu.fi`. There is no installation procedure. To start using the library, you just include some files. In order to keep the core of the library clean, and to allow users to selectively take on the features of the LL, the basic include file `ll.hpp` just includes the lambda functors, bind functions and the basic operators. Control constructs and other extras can be included selectively. The file `ll_all.hpp` has everything in it for users who want it all.

The library is currently in the namespace `"boost"`. There are a couple of defines to help manage the namespace issues, one which allows you to change the LL namespace, another to remove namespacing all together.

Part II
Implementing the Lambda Library

5 General library design

The LL consists of a number of layers. Each layer implements a certain task orthogonal to the tasks of the other layers. This orthogonal layering is the key to manage the variation between different types of lambda expressions. Within each layer, it is enough to write template specializations with respect to a single varying factor.

Even with the strict layering, the architecture of the LL is complex. We will not try to describe the library code entirely, but rather will look at the chain of template instantiations taking place while compiling a simple lambda expression. In this way, the purpose of each layer becomes clear.

The presented code is somewhat simplified for clarity. Particularly, while the arguments of the lambda expressions are always stored inside a *tuple* object (see section 5.2) in the lambda functor, the details vary. Depending on the lambda expression and the type of the arguments, the arguments are stored either as const or non-const, and either as copies or references. This variability is achieved with a set of traits classes which perform the necessary type conversions. Therefore, in the library code each argument type is wrapped inside one of these traits class instantiations. We do not show these wrappings here. Moreover, we commonly use the `make_tuple` functions (see section 5.2) to create the argument tuples, while in the library code we are calling the tuple constructors directly and specify the full type of the tuples. Although these details are crucial for a fully functional library, we believe that omitting them here is beneficial: the code is much easier to read and the general library design does not get lost as easily.

To start with, the central template in the implementation is the template class `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 `free1 + 1` and `bind(foo, 1, free2, free1)`, exception handling lambda functors etc. are all instances of the `lambda_functor` template.

5.1 Lambda expressions as partial function applications

The LL implementation is best understood by viewing lambda expressions as delayed and partial function applications. In a partial function application, a function is called with empty slots, placeholders, in the argument list. Hence, some formal parameters of the function are bound normally to the actual arguments, while some parameters are left open. The result of such a partial function call is another function with possibly fewer (or more, see below) arguments. When this new function is called with the arguments that were left open, the original function is called with an argument list where these arguments have been substituted for the placeholders. For example, the expression `free1 = 1` is a partial application of the binary assignment function, where the first argument is left open and the second argument is bound to 1. When this unary partially applied function is called with some argument `x`, `free1` gets replaced and the original `operator=` function is called as `x = 1`.

The model of partial function application must be generalized a bit in order to completely describe the functionality of the LL: in addition to a constant value, an argument can be bound to another partially applied function. For example, in the expression `-(free1 * free2)` the argument of the unary `operator-` function is bound to a partially applied function `free1 * free2`. Note that in this example, the partial application of the unary `operator-` results in a binary function. Furthermore, in the LL a zero-argument (see the example in section 2.1) function can be a target of a partial application. This means that the function call is simply delayed.

5.2 The role of tuples

The parameters of the `lambda_functor` template describe the type of the delayed operator/function and the types of the bound arguments and placeholders. Although the number of bound and unbound arguments can vary, the `lambda_functor` parameterization covers all cases (there are a few specializations for `lambda_functor` though). This is possible because of *tuples*. We wrap the parameters inside a tuple object and can thus treat the parameter list as one entity, regardless of the true number of parameters. 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 functions. However, thanks to tuples this code repetition can be kept to a minimum.

In short, a tuple 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:

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

Hence, value types, const and non-const types, reference types, other tuples, function pointers etc. are valid element types. With the exception of some unavoidable restrictions (e.g. non-copyable types cannot be stored in a tuple except as references), any C++ type can be used as a tuple element type.[4]

Tuples can be constructed explicitly:

```
tuple<int, double, A>(1, 3.14, A())
```

or with the `make_tuple` function (cf. `std::make_pair`):

```
make_tuple(1, 3.14, A())
```

The `N`th element of a tuple, where `N` is an integral constant, can be accessed with no runtime overhead using the `get` function as `get<N>(x)` or `x.get<N>()`. The two alternatives are interchangeable. The type of a tuple element can be queried with the traits class `tuple_element_type`. The type expression:

```
tuple_element_type<N, T>::type
```

retrieves the type of the `N`th element of a tuple of type `T`.

To use the LL, you do not have to know anything about tuples. To understand the implementation, the above should be enough. For an interested reader, tuples are described in greater depth in [Jär01, Jär99a, Jär99b].

Note that our current implementation of tuples has an upper limit of 10 arguments. When we mention a limit on the number of cases in the LL as being 10 or less, this is where the limitation originates. We could easily increase this number but for now it has been a reasonable upper limit.

[4] Our current implementation does not support volatile-qualified types.

5.3 Placeholders

In the first part of the article we went over how to use placeholders. Here we explain the internals of what they are. Placeholders in the LL are just instances of a templated class `placeholder` wrapped in a `lambda_functor`:

```
template <int I> class placeholder {};

typedef const lambda_functor<placeholder<FIRST> >  free1_type;
typedef const lambda_functor<placeholder<SECOND> > free2_type;
typedef const lambda_functor<placeholder<THIRD> >  free3_type;
```

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 constants `FIRST`, `SECOND` and `THIRD` are propagated from placeholder types to more complex lambda functors and carry information about the arity of the lambda functors. Consequently, we call them *arity codes* (see section 5.3.1).

Notice that the placeholder has no knowledge of the type of the argument it stands for. As we explained in section 2.2, 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 substitutions have been performed. This is possible, since lambda functor templates define the `operator()` functions as member templates. The argument types are not known until the functor is actually called and implicit instantiation of the `operator()` member template is performed.

When the compiler has deduced these argument types, two recursively alternating deduction chains commence. The compiler must deduce the return type of the `operator()`. This is performed with the return type deduction templates of the LL. As input, the return type deduction system takes the underlying operator, i.e. the target function, of the lambda functor together with the types of the arguments this operator will be called with. In order to figure out the argument types, the compiler may have to resort to the return type deduction system recursively. We'll first demonstrate a simple non-recursive case:

```
int x;
  ...
(free1 = 1)(x);
```

First, the compiler deduces the argument types to the underlying assignment operator; the instantiation becomes `operator=(int&, const int&)`. Next, the return type is deduced based on the information that we are performing an assignment operator between arguments of type `int`.

Function compositions cause a bit more of a headache (for the implementers of the library, not for the user). For example, consider the lambda expression, `free1 + (free1 * free2)`. 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 it 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. As complex as this is, the LL correctly deduces return types for such compositions. The return type deduction system is explained in more details in section 6.7.

5.3.1 Arity codes

Each lambda functor has an arity code as one of its template arguments. When a lambda expression is evaluated, the resulting lambda functor gets the highest arity code from its subexpressions. In other words, the arity code of a lambda functor states the highest placheholder index that is present in any of the subexpressions. For example:

```
-free1                 // arity code = FIRST
free1 - free2          // arity code = SECOND
free1 - free2 - free3  // arity code = THIRD
```

There is a direct mapping from the arity codes to the actual arities of the lambda functors: `FIRST` means a unary functor, `SECOND` a binary functor etc. Note particularly, that even in the absence of placeholders with a lower arity code, the placeholder with the highest arity code determines the arity of the lambda functor. For example, the lambda expression `&free3` creates a 3-argument lambda functor. When called, this lambda functor discards the first two arguments and returns the address of the third argument.

Note that arity code can have the value `NONE` as well, which means that the lambda functor is nullary, and the value `EXCEPTION`, which indicates that the `freeE` placeholder was used in a lambda expression.[5]

6 A walk through the layers

We have now covered the basics and can guide you through the different layers of the LL — with the simple lambda expression we promised. Our example lambda expression is `free1 < 0`. This expression creates a unary lambda functor, which compares its argument to zero.

[5]There is still one more constant (`RETHROW`), which can be bitwise ored with other arity codes. It is used internally to guarantee that a `rethrow` lambda expression is only used within a catch block.

6.1 Overloaded operators for lambda expressions

For each binary operator `op`, there are three overloaded definitions:

```
lambda_functor op any_type
any_type       op lambda_functor
lambda_functor op lambda_functor
```

For each unary operator, one overloaded definition is enough.

When the compiler encounters our example expression, the function template shown in figure 2. is selected as a result of the overload resolution. This operator takes a `lambda_functor` as the left-hand argument and constructs another `lambda_functor`. 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_args<
    action<2, relational_action<less_action> >,
    tuple<lambda_functor<placeholder<FIRST> >, const int>,
    FIRST
  >
>
```

This type carries information about the invoked operator, the types of the arguments to the operator, and the arity of the lambda functor. The type `action<2, relational_action<less_action> >` means that the lambda functor stands for the binary less than operator (see section 6.4 about *action classes*). The arguments `free1` and `0` are stored in a tuple, which gets the type `tuple<lambda_functor<placeholder<FIRST> >, const int>`. The `dig_arity` is a template which extracts the arity code of a lambda functor. The arity code of `free1` is `FIRST` (see section 5.3.1). As the constant `0` has no arity code, the value `FIRST` is propagated to the resulting lambda functor, stating that its function call operator is unary.

Note that we need three template arguments to store the above pieces of information. The `lambda_functor_args` is an intermediate template that groups these three parameters into a single class. Due to this arrangement the `lambda_functor` template has only one template parameter, which makes overloading operators and functions for lambda functors simpler.

6.2 The `lambda_functor` template

The `lambda_functor` templates define the function call operators that are called from STL algorithms. There are four `lambda_functor` specializations for different arities (0-3 arguments). In addition to this, `lambda_functor` has a common specialization for the three placeholder types and one more

```
template<class Arg, class B>
inline const
  lambda_functor<
    lambda_functor_args<
      action<2, relational_action<less_action> >,
      tuple<lambda_functor<Arg>, const B>,
      dig_arity<Arg>::value
    >
  >
operator< (const lambda_functor<Arg>& a, const B& b)
{
  return lambda_functor<
           lambda_functor_args<
             action<2, relational_action<less_action> >,
             tuple<lambda_functor<Arg>, const B>
             dig_arity<Arg>::value
           >
         > ( make_tuple(a, b) );
}
```

Figure 2: The `operator<` overloaded for a lambda expression with a lambda functor as the left-hand argument.

```
template <class Action, class Args>
class lambda_functor<lambda_functor_args<Action, Args, FIRST> >
  : public lambda_functor_base<Action, Args> {
public:

  explicit lambda_functor(const Args& args) :
    lambda_functor_base<Action, Args>(args) {}

  template<class A>
  typename return_type<lambda_functor, A&, nil, nil>::type
  operator()(A& a) const {
     return this->template call<
       typename return_type<lambda_functor, A&, nil, nil>::type
     > (a, const_nil(), const_nil());
  }

  template<class RET, class A>
  RET ret_call(A& a) const {
    return
      this->template call<RET>(a, const_nil(), const_nil());
  }
};
```

Figure 3: The specialization for unary lambda functors. The `template` keyword before the `call` invocation is required by the C++ standard; it helps the parser to figure out that `call` is a templated member function rather than a data member.

which is used with delayed exception handling for a combined total of 6 partial specializations.

Since our example lambda expression has one unbound parameter, and the arity code is thus `FIRST`, a `lambda_functor` specialization with a unary `operator()` template is instantiated (figure 3.). The task of the `operator()` function template is to initiate the return type deduction chain and forward the call further. The return type deduction traits, as well as the functions that substitute the placeholders for the actual arguments are only implemented for the case of 3-argument lambda functors. Hence, in this unary case the second and third arguments are filled with special *nil*-objects.

Suppose that the lambda functor in figure 3. is called with an argument of type `int`. For example:

```
int i; ... ; (free1 < 0)(i);
```

Now the template parameter `A` becomes `int` and the

```
return_type<lambda_functor, int&, nil, nil>::type
```

can be resolved. We'll consider the return type deduction traits later in section 6.7, just assume for now that the return type is correctly deduced to `bool`, or actually to const bool.[6]

After filling the missing arguments with nil-objects (`const_nil()`) does this), all arguments are forwarded to the `call` function defined in the base class `lambda_functor_base`.

In addition to the function call operator, the `lambda_functor` templates define another member function. This `ret_call` function has the same functionality as the function call operator, except for the return type which is given as an explicitly specified template argument, rather than deduced with the return type deduction traits. The LL uses `ret_call` in internal calls to lambda functors to prevent the compiler from doing the same return type deductions repeatedly.

6.3 The `lambda_functor_base` template

The `lambda_functor_base` templates store the argument tuple and define the `call` functions which perform the argument substitution. The template is specialized for each different length of function argument list; while the varying factor in `lambda_functor` specializations was the arity of the lambda functor, here it is the arity of the target function.

Our example target function, `operator<`, is binary. Therefore the specialization shown in figure 4. is selected. As a result of the call from the `lambda_functor::operator()` function, the `call` member template gets instantiated as:

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

This function delegates the argument substitution to the `select` functions (see section 6.5) and forwards the substituted argument list to `Act::apply` function; in this case to `relational_action<less_action>::apply`.[7]

6.4 Action classes

Action classes are wrappers for the target functions. There is one for each overloaded operator (`less_action`, `minus_action` etc.). There are also action classes for different types of functions: function pointers, pointers to

[6]In order compose lambda functors we return temporaries as const types. This allows us to pass the result of one lambda functor to another lambda functor as a reference argument.

[7]Note that the return type deduction is only performed in the lambda functor's `operator()`. The deduced return type is then forwarded as an explicitly specified template parameter both to the `lambda_functor_base::call` and further to the `Act::apply` function. This arrangement lessens the load of the compiler significantly.

```
template<class Act, class Args>
class lambda_functor_base<action<2, Act>, Args> {
  Args args; // the argument tuple
public:
  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 Act::template apply<RET>(
      select(get<1>(args), a, b, c),  // select substitutes
      select(get<2>(args), a, b, c)); // the arguments
  }
}
```

Figure 4: The `lambda_functor_base` specialization for unary functors.

member functions etc. The action class is part of the type of the lambda functor instantiation and its sole purpose is to provide a static member function for calling the target function.

The action classes are separated into groups. The groups are defined by similar operation, for instance, arithmetic, logical, boolean, void, etc. This grouping allows us to implement the return type deduction rules for sets of operators. For instance, all arithmetic operators follow the integral promotion rules, while the return type of all relational actions is `const bool`. Here is the code for the `less_action` class:

```
class less_action {};
template <class ActionClass> relational_action;

// The specialization of relational_action for less than
struct relational_action<less_action> {
 template<class RET, class A, class B>
   RET static apply(A &a, B &b) { return a < b; }
};
```

The `apply` function eventually calls the target function, which in our example is the built-in `operator<`.

6.5 Argument substitution

The apply function is called with a merged argument list, where the arguments have already been substituted for the placeholders. The `select` functions called from `lambda_functor_base` templates perform this substitution. The call

```
select(get<1>(args), a, b, c)
```

selects either the bound argument from the `args` tuple (`get<1>(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. In our example, the following specialization for the `free1` placeholder overrides the default behavior by selecting the first parameter of the `lambda_functor_base::call` function:

```
template<class A, class B, class C>
inline A&
select(const lambda_functor<placeholder<FIRST> >&,
       A& a, B&, C&) {
  return a;
}
```

Analogous overloaded functions exist for all placeholder types.

Function decomposition is also achieved with overloading select functions. Instead of just returning one of the arguments, the select functions call the lambda functor with some of the other arguments, and return the result of that call. We do not go into details, but the following pseudo code shows the general idea. This specialization would match for binary lambda functors. Notice the call to `ret_call` rather than the function call operator (see section 6.2):

```
template<class RET, class A, class B, class C>
inline RET
select(const binary_lambda_functor& op, A& a, B&, C&) {
  return op.template ret_call<RET>(a, b);
}
```

6.6 Intermediate 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_args<
       action<2, relational_action<less_action> >,
       tuple<placeholder<FIRST>, const int>,
       FIRST
     >
   >
   ```

2. The unary function call operator of this class is called with the variable `i` of type `int`. The return type is resolved to `const bool` with the instantiation `return_type<lambda_functor, int&, nil, nil>`. The return type and the arguments are forwarded to the `call` function of the base class:

   ```
   lambda_functor_base<
     action<2,relational_action<less_action> >,
     tuple<lambda_functor<placeholder<FIRST>, const int>
   >::call<const bool>(i, const_nil(), const_nil());
   ```

3. The `call` function calls the `select` function to choose between the bound arguments stored in the tuple member and the argument `i` supplied as a parameter:

   - The first argument: `select(get<1>(args), a, b, c)` turns into `select(free1, i, const_nil(), const_nil())` which returns a reference to `i`.
   - The second argument: `select(get<2>(args, a, b, c)` becomes `select(0, i, const_nil(), const_nil())` and returns `0`.

4. The return type is deduced and the function

   ```
   relational_action<less_action>::apply<const bool>(i, 0)
   ```

   is called, which returns the result of the expression `i < 0` all the way to the caller of the `lambda_functor::operator()`.

```
template<class Act, class Args, int Code,
         class A, class B, class C>
struct return_type<lambda_functor<action<2, Act>, Args, Code>,
                   A, B, C> {
  typedef typename
    return_type<typename tuple_element_type<1, Args>::type,
                A, B, C>::type X;

  typedef typename
    return_type<typename tuple_element_type<2, Args>::type,
                A, B, C>::type Y;

  typedef typename return_type_2<Act, X, Y>::type type;
};
```

Figure 5: The binary function case of the topmost layer of the return type deduction system.

6.7 Return type deduction

The `return_type` template constitutes the topmost layer of the return type deduction system. It takes four template parameters: a lambda functor and the types of the three actual arguments of the lambda functor. The `return_type` template is specialized with respect to the arity of the underlying action (1, 2, 3 and N argument cases). In our example, the action is binary and thus the specialization in figure 5. is instantiated.

The template parameter `Act` is the action class type and the `Args` tuple type contains the bound argument and placeholder types. `A`, `B` and `C` are the types of the parameters to substitute for the placeholders. The definition can be interpreted as follows: Use the traits class `return_type_2<Act, X, Y>` to deduce the return type of performing the binary action `Act` to some parameters of types `X` and `Y`. The parameters `X` and `Y` are obtained recursively with the `return_type` template.

In our example, the action class is `relational_action<less_action>`, the first tuple element type is `lambda_functor<placeholder<FIRST> >` and the second element type is `const int`. Hence, the specialization:

```
template <class A, class B, class C>
struct
  return_type<lambda_functor<placeholder<FIRST> >, A, B, C> {
    typedef A type;
};
```

is used to deduce that `X` equals to `int`. `Y` is deduced to `const int` with the primary template:

```
template <class Arg, class A, class B, class C>
struct return_type {
  typedef Arg type;
};
```

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

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

Above we stated that the return type deduction system is easily extendible. This is achieved by adding specializations for the `return_type_2` template. For example, the following specialization defines that the return type of relational operators for a user-defined type `FuzzyNumber` would be `FuzzyTruthValue`:

```
template<class Act>
struct return_type_2<relational_action<Act>,
                     FuzzyNumber, FuzzyNumber> {
  typedef const FuzzyTruthValue type;
};
```

Furthermore, the LL has a special type promotion scheme for numeric types, modeled after the one used in the Blitz library [Bli99]. In this scheme each type is given a value. The value of the result type of an arithmetic operation is the largest of the parameter values, i.e. if `short` = 300 and `long` = 500, then `short op long` is max(300,500) or 500, or `long`.

User defined types are added to the promotion scheme by assigning them a value. For example, the result type of the addition of a `FuzzyNumber` and any standard numeric type can be defined with the following template. Here `FuzzyNumber op` *any native type* is promoted to `FuzzyNumber`; `value` is large enough to get the correct promotion weight.

```
template<> struct promote_code<FuzzyNumber> {
  static const int value = 2000;
};
```

6.7.1 Overriding return type deduction

For one-time-only uses it may be an overkill to define the return type deduction traits for some user-defined types. As we stated, the return type deduction system can be overridden and the return type specified on the fly within a lambda expression. This is done by wrapping the lambda expression inside a `ret<T>` function, where `T` is the return type. For example, had we not defined the above traits for `FuzzyNumber` classes, we could write as follows:

```
vector<FuzzyNumber> fuzzy_numbers;
vector<FuzzyTruthValue> fuzzy_bools;
FuzzyNumber x;
  ...
transform(fuzzy_numbers.begin(), fuzzy_numbers.end(),
          fuzzy_bools.begin(),
            ret<FuzzyTruthValue>(x < free1));
```

Note that `ret<T>` does not call any of the C++ cast statements. It merely defines the return type for a given lambda expression. The true return type of the target expression must be compatible with `T`, that is, implicitly convertible to `T`.

6.8 Bind expressions

In the preceding sections, we used the expression `free1 < 0` to walk through the central parts of the implementation. Our example didn't cover the implementation of the `bind` functions, which is the topic of this section.

The section 2.3 showed several examples of bind expressions. A bind expression is a call to one of the overloaded `bind` functions, where the first argument is the target function and the remaining ones are used as the argument list, after substitutions, to the target function when it is called.

To begin with, the `bind` function is overloaded to take any number of arguments between zero and 10. Further, for each argument list length we need four overloaded definitions to cover the cases of non-member function pointers, pointers to const and non-const member functions and arbitrary function objects as target functions. As an example, the figure 6. shows the 3-argument `bind` function that matches `bind` invocations, where the target function is a pointer to a binary non-member function. The following bind expression is an example of a call that matches this `bind` function.

```
int foo(float a, int b);
  ...
bind(&foo, free1, 1);
```

The action class of the resulting lambda functor is `function_action<3>`. Compared to, say, the `less_action`, function action classes have one extra layer to unify the syntax of calling different types of functions (member vs. non-member) before the actual `apply` function is called. Note that although the target function is binary, the arity of the action class is 3, since the action class treats the target function as a normal substitutable argument. The `bind` function groups the target function with the other arguments into a tuple which is stored in the lambda functor.

Note the type expression `combine_arities<Arg1, Arg2>::value`. In the above example case the target function is a pointer to a function, and thus not a lambda functor. However, the remaining two arguments to the `bind` function can be anything, particularly they can be lambda functors. The `combine_arities` traits class computes the arity code of the lambda functor type to be created from the arity codes of the argument types; slightly simplified it selects the highest arity code of its argument types.

We need the `bind` functions to implement the interface for creating lambda functors from various types of functions, and the `function_action` templates to implement the calling rules for these functions. Further, we need some templates for the return type deduction system. However, ascribe to the orthogonal layers in the LL, these are the only components that are specific to bind expressions. Other template layers are not dependent on the type of the target function in the lambda functor.

This concludes our discussion about the implementation of the basic features of the Lambda Library.

7 Special cases in operator overloading

7.1 Pointer to member operator

In section 3.2 we stated that the argument of the `operator->*`, the *member pointer*, can refer to either a member object or a member function. Consequently, the global `operator->*` is overloaded differently for these cases. If the member pointer is an object, we return a lambda functor with a reference to that object. If the member pointer is a pointer to a function, the C++ language doesn't give much freedom what to do next. The result is something like a pending member function call; the only thing that can be done with the result is to pass it a list of arguments (see [Mey99] for an in-depth analysis of overloading `operator->*`). Hence, we return a temporary object, which has the function call operator defined for the same number of arguments as does the member pointer function. This `member_ptr_lambda_functor` serves as a proxy. It is not a lambda functor that gets passed to an STL algorithm, but rather the function call operator of the `member_ptr_lambda_functor` creates and returns that lambda functor.

In section 3.2 we showed the call `free1->*(&data::foo)` to the unary

```
template <class Par1, class Par2, class Result,
          class Arg1, class Arg2>
inline const
lambda_functor<
  lambda_functor_args<
    action<3, function_action<3> >,
    tuple<Result (*)(Par1, Par2), const Arg1, const Arg2>,
    combine_arities<Arg1, Arg2>::value
  >
>
bind(Result (* const & f)(Par1, Par2),
             const Arg1& a1, const Arg2& a2) {
  return
    lambda_functor<
      lambda_functor_args<
        action<3, function_action<3> >,
        tuple<Result (*)(Par1, Par2), const Arg1, const Arg2>,
        combine_arities<Arg1, Arg2>::value
      >
    > (make_tuple(f, a1, a2));
};
```

Figure 6: The `bind` function for the case where the target function is a pointer to a binary non-member function.

```
template<class Arg, class Result, class Object, class Arg1>
inline const
  member_ptr_lambda_functor
    <1, Arg, Result (Object::*)(Arg1) const, Arg1>
operator->*(const lambda_functor<Arg>& a,
            Result(Object::* const & f)(Arg1))
{
  return member_ptr_lambda_functor
           <1, Arg, Result(Object::*)(Arg1), Arg1>(f, a);
}
```

Figure 7: Member pointer operator for unary non-const member functions.

member function `int data::foo(int)`. The figure 7 shows the overloaded definition of `operator->*` which gets instantiated in this case. The creation of the `member_ptr_lambda_functor` proxy in this function triggers the instantiation of the specialization shown in figure 8. We do not show the code of the inherited template class `member_ptr_base`; it is just a holder of the arguments, the lambda functor and the member pointer, passed to the overloaded `operator->*`.

In the example of section 3.2 we passed the constant `5` to the result of the `operator->*` call with the expression `(free1->*(&data::foo))(5)`. This is a call to the `operator()` of the `member_ptr_lambda_functor`, and it results in the lambda functor that can be passed to an STL algorithm.

Furthermore, we stated that the same functionality can be achieved using the `bind` function. Indeed, the arguments are bound into a tuple the same way that `bind` does it and the action class is `function_action`. This is the same action that is used by `bind`. It does not make a difference which syntax is used; both interfaces create the exact same lambda functor instantiation.

7.2 comma, && and || operators

The comma operator and the logical *and* and *or* operators follow the C++ rules for argument evaluation order and short-circuiting. Instead of implementing these operators with an action class like `less_action` and calling the action class from the `lambda_functor_base` template, we call the operator directly from `lambda_functor_base`. For this, we need to specialize the `lambda_functor_base` for each of these operators. This guarantees that only the necessary expressions will be evaluated and in the correct order. Another reason for implementing the comma operator in this way is that the built-in comma operator accepts a `void` argument. We cannot call a function, particularly the `apply` function of an action class, with an argument of type `void`. Therefore we must call the comma operator directly.

As an example, the specialization of the `lambda_functor_base` for the logical *and* operator is shown in figure 9. Note that the `operator&&` is called directly in the `call` function of `lambda_functor_base`. The `&&` calling rules apply; the right-hand `select` function is never called if the left-hand `select` returns false.

7.3 Conditional expression

The conditional expression `?:` is not allowed to be overloaded. Besides this, the result type rules for this operator are very complex (see [C++98, section 5.16]). Nevertheless, the LL counterpart for this operator that we showed in section 3.4, the `if_then_else_return` function template, follows these rules to the extent that it is possible. Rather than spend time discussing how this bit of the library works, we are going to skip it. The return type

```
template<class Arg, class ObjectMemPtr, class Arg1>
class member_ptr_lambda_functor<1, Arg, ObjectMemPtr, Arg1> :
  private member_ptr_base<Arg, ObjectMemPtr> {
public:
  typedef member_ptr_base<Arg ObjectMemPtr> inherited;

  member_ptr_lambda_functor(ObjectMemPtr const &f,
                            const lambda_functor<Arg> &a) :
      inherited(f,a) {}

  lambda_functor<
    lambda_functor_args<
      action<3, function_action<3> >,
      tuple<ObjectMemPtr, lambda_functor<Arg>, const Arg1>,
      combine_arites<Arg, Arg1>::value
    >
  >
  operator()(const Arg1 &a1) const {
    return
      lambda_functor<
        lambda_functor_args<
          action<3, function_action<3> >,
          tuple<ObjectMemPtr, lambda_functor<Arg>, const Arg1>,
          combine_arites<Arg, Arg1>::value
        >
      >
      (
        make_tuple(inherited::m_memPtr,
                   inherited::m_lambdafunctor,
                   a1)
      );
  }
};
```

Figure 8: The `member_ptr_lambda_functor` specialization for unary non-const member functions.

```
template<class Args>
class lambda_functor_base<
        action<2, logical_action<and_action> >, Args
      > {
  Args args;
public:
  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<1>(args), a, b, c) &&
           select(get<2>(args), a, b, c);
  }
};
```

Figure 9: `lambda_functor_base` template specialization for the logical *and* operator.

deduction is complex and only narrowly interesting as a technique. We invite the interested reader to read the library code directly.

8 Control constructs

To be able to provide the exact same functionality, that is the exact same argument evaluation order, to the underlying C++ control constructs, the LL control constructs are again specializations of the `lambda_functor_base` class.

As an example, we show the `lambda_functor_base` specialization for the `for_loop` in figure 10. Note that the return type is a specialization to return `void`, therefore no return statement is necessary. Since the `select` functions are written inside the for loop, the code will obey the regular C++ rules of a for loop. All loop controls in the LL are implemented analogously.

8.1 Switch as a lambda expression

For the switch statement we go into a bit more detail. The switch lambda expression is a composition of several lambda expressions, and we show the techniques to ensure that the component lambda expressions are of the right kind. It is a bit hairy but we try our best to explain the whys and the whats.

To implement the switch lambda expression we have specialized the `switch_statement` function for up to 9 case statements. These functions

```
template<class Args>
class lambda_functor_base<
        action<4, return_void_action<forloop_action> >, Args
      > {
  Args args;
public:
  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 {
    for(select(get<1>(args), a, b, c);
        select(get<2>(args), a, b, c);
        select(get<3>(args), a, b, c))

      select(get<4>(args), a, b, c);
  }
};
```

Figure 10: `lambda_functor_base` specialization for the for loop.

return a `lambda_functor` with a `switch_action` as the action class, and a tuple holding the different cases. Also the `lambda_functor_base` templates are specialized differently for each number of case statements. Actually, there are two specializations for each such case: one with the last case being the default statement, and one without the default statement. The underlying switch statement code, the one that calls the built-in switch statement, is built using one of these specializations.

Again, we use a simple example to walk through the different parts of the implementation. Consider the following switch lambda expression consisting of a single case and a default:

```
switch_statement(free1,                            // the test
                 case_statement<0>(free1 = 100),   // case 1:
                 default_statement(free1--));      // default:
```

The different case statements and the default statement are lambda expressions, but they would not make any sense outside of a switch statement. Further, the switch lambda expression would not make any sense, if instead of case lambda expressions, the arguments would be arbitrary lambda expressions. Consequently, we have defined a `switcher` class, which is a lambda functor that can only be used as a case of a `switch_statement`. The `switcher` class inherits from a `lambda_functor` instantiation, thus `switcher` has all the properties of a lambda functor. In order to reduce the number of special cases, the same template covers both regular cases and the default case. A template parameter is used to indicate which of these types the case statement is:

```
enum CaseEnumType { CASE, DEFAULT };

template<CaseEnumType caseType, int CaseValue, class Args>
class switcher : public lambda_functor<Args> {
public:
 template<class T>
   switcher(const lambda_functor<T>& a)
    : lambda_functor<Args>(a) {}
};
```

This next template is a function for setting up the switcher class for a regular case statement. Notice that the `CaseValue` template parameter is not deducible; it is the case value that the user specifies explicitly within the call to this function:

```
template<int CaseValue, class Arg>
const switcher<CASE, CaseValue, lambda_functor<Arg> >
case_statement(const lambda_functor<Arg> &arg) {
  return switcher<CASE, CaseValue, lambda_functor<Arg> >(a);
}
```

The function for setting up the switcher class for a default case statement is similar, except for the different case type parameter (`DEFAULT`), and for the fact that the case value is already set to a known value (`0` is an arbitrary choice):

```
template<class Arg>
const switcher<DEFAULT, 0, lambda_functor<Arg> >
default_statement(const lambda_functor<Arg> &arg) {
  return switcher<DEFAULT, 0, lambda_functor<Arg> >(a);
}
```

In order to combine multiple switcher objects into one object we have created a template to hold a group of them. This class allows us to treat all variations of the switch as one type for return type deduction purposes, the advantage being that we already have a `return_void_type` class that we can reuse:

```
template<class case1, class case2 = nil, class case3 = nil,
         ... , class case 9 = nil>
class switch_list_action;
```

```
template <class TestArg, int Case1, class SwitchAct1,
          CaseEnumType CaseType2, int Case2, class SwitchAct2>
inline const
lambda_functor<
  lambda_functor_args<
    action<3, return_void_action<
                switch_list_action<
                  switcher<CASE, Case1, SwitchAct1>,
                  switcher<CaseType2, Case2, SwitchAct2>
                >
              >
    >,
    tuple<lambda_functor<TestArg>, SwitchAct1, SwitchAct2>,
    combine_arities<TestArg, SwitchAct1, SwitchAct2>::value
  >
>
switch_statement(
  const lambda_functor<TestArg>& a1,
  const switcher<CASE, Case1, SwitchAct1>& a2,
  const switcher<CaseType2, Case2, SwitchAct2>& a3)
{
  return
    lambda_functor<
      lambda_functor_args<
        action<3, return_void_action<
                    switch_list_action<
                      switcher<CASE, Case1, SwitchAct1>,
                      switcher<CaseType2, Case2, SwitchAct2>
                    >
                  >
        >,
        tuple<lambda_functor<TestArg>,
              SwitchAct1, SwitchAct2>,
        combine_arities<TestArg,
                        SwitchAct1, SwitchAct2>::value
      >
    >
    (make_tuple(a1, a2, a3));
}
```

Figure 11: The switch_statement.

Next, the overloaded switch_statement function for two cases is shown in figure 11. The function constructs the complex lambda_functor which contains a tuple of the arguments, and encodes the underlying C++ switch statement within its template arguments. The use of return_type_void indicates that switch control lambda functors have no return type. The template combine_arities looks at the three lambda functors (the switch expression part, the first case statement and the default case) for the highest arity code and sets the arity code to that number (see section 6.8). This arity code determines which specialization of the lambda_functor template we will use. In this case the arity is FIRST, which means we can use the same lambda_functor class as we used for the example expression free1 > 0 (see section 6).

Notice that although we use the generic switcher class as two of the arguments, we do not allow a default statement in any place but the last argument. Furthermore, we do not allow any other lambda expression types as switchers. Note also that the switchers are stored into the argument tuple of the resulting lambda expression as normal lambda functors; the fact that they really are, or were, switchers is deliberately lost at this point. The purpose of the switcher template is to guarantee that the switch lambda expressions are composed of the right kinds of components. When the lambda functor for the switch statement has been constructed, this has been guaranteed and the switchers are not needed anymore (they do prevail in the action class type, however).

The instantiation of the lambda_functor class triggers the search for a matching lambda_functor_base. The LL has a set of specializations of the lambda_functor_base for switch statements. The compiler picks one of these as the closest match and continues the instantiation. The figure 12. shows the specialization of lambda_functor_base for a switch statement with one case statement and the default case. This code expands similarly to the for loop, with the select calls looking for placeholders and the return type RET being set to void. The end result is a regular C++ switch statement.[8]

We used a different template to solve each layer of the problem. By using all these templates we have reached a solution, where the size of the code is linear, rather than exponential, with respect to the number of case statements we support. We can handle any combination of placeholders, and any lambda expression, even another switch lambda expression, as part of the switch lambda expression.

The do_once implementation is analogous to the switch_statement implementation: we have 9 overloaded do_once function templates, and another 9 specializations of the lambda_functor_base, one for each number of cases.

[8]Whew! What a lot of templates just to get back to plain old C++ code.

```
template<class Args, int SwitchCase1, class SwitchArg1,
         class SwitchArg2>
class lambda_functor_base<
        action<3, return_void_action<
                    switch_list_action<
                      switcher<CASE, SwitchCase1, SwitchArg1>,
                      switcher<DEFAULT, 0, SwitchArg2>
                    >
                  >
        >,
        Args
      >
{
  Args args;
public:

  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 {
        switch(select(get<1>(args), a, b, c))
        {
          case SwitchCase1:
            select(get<2>(args), a, b, c); break;
          default:
            select(get<3>(args), a, b, c); break;
        }
  }
};
```

Figure 12: The `lambda_functor_base` specialization for a switch statement with one case statement and the default statement.

9 Delayed exception handling

The exception handling lambda expression consists of one try block lambda expression and potentially many catch block lambda expressions. The catch block can either be for a specific exception type, or for any exception type (cf. the `catch(...)` statement), if it is the last catch block. Hence, the basic structure is the same as in the switch lambda expression.

However, while the return type of a switch lambda functor is void, the exception handling lambda functor can return a value.[9] This makes things difficult, since the lambda functor can return either from the try part or from one of the handlers. A try block lambda functor can either return an argument, throw or return `void`, and a catch block lambda functor can throw, or rethrow, or exit normally and return a value, or `void`. In all these combinations, the return types must match. In the case of a normal exit from a catch block, the catch block lambda functor must return a type which can be converted to the same type as the return type of the try block lambda functor. On the other hand, if the catch block always throws, or rethrows, the return types can be different as the catch block return type is never used.

Our first attempt to cover all these different cases had six specializations for one catch block, for the two catch blocks case we had 12 specializations, the next 24 and we stopped.[10] We added some extra template layers to check statically whether the lambda functor in a catch block will throw or not and got rid of the combinatorial explosion of the template specializations. Currently there are two cases per number of catch blocks, one in which the last catch is a `catch(Type&)`, and the other in which it is a `catch(...)`.

Rather than go through the whole gory code we are going to wave our hands and just touch on the highlights. If you are really interested we suggest that you get the source and take a look. Again the implementation relies on specializations: two `lambda_functor_base` specializations for each number of catch blocks. There is a `catch_exception` function that returns a template `catcher`, which is similar to the template `switcher`. Each `catcher` holds a catch block lambda functor.

The `try_catch` functions are similar to the `switch_statement` functions. The first argument is an arbitrary lambda functor, and the following arguments are `catcher` templates. Only the last argument can be a `catch_all`.

The specializations of the `lambda_functor_base` are similar to the specializations for the `switch_statement`, with one extra template layer in the `call` functions. There is again an upper limit of 9 catch lambda expressions because of the limit of the number of arguments to the tuples: 9 exception handlers seems more than adequate.

We take a glimpse of the code with a simple example: The following code

[9] Hence, an exception handling lambda expression can be used just as any lambda expression, i.e. as a subexpression of another lambda expression.

[10] Ok, we didn't but we should have, 48 specializations was really ridiculous.

calls a function `foo` on every element in some container `v`. If `foo` throws an exception of type `exp_type`, it gets caught and a message is sent to `cout`:

```
void foo(int); // our function foo.
for_each(v.begin(),v.end(),
           try_catch(
             bind(foo, free),
             catch_exception<ExpType>(cout << "Caught it.")
           )
        );
```

The code expands to use the specialization of the `lambda_functor_base` shown in figure 13. Note the extra layer in the `call` member function of the catch block. Instead of calling the `select` functions directly, we call another `call` function[11], which is a static member of the `return_or_throw` template. This template calls the `select` functions, and handles the two possible outcomes: either the code will throw again, or it must return an argument that can be converted to type `RET`.

Notice that inside of the catch block the third argument to the `call` function is the exception (the parameter `e`) that was thrown, not the third argument passed into the try block (the parameter `c`). The client refers to the third argument with `free3` and to the exception object `freeE` as is explained in section 3.8. The exception placeholder takes the place of the `free3` placeholder and thus reuses the machinery for placeholder substitutions. This arrangement is the reason why the `free3` placeholder cannot be used within a catch block.

10 Performance testing

The preceding sections show how the LL works, and how it is implemented. In this section we focus on the performance issues. The news is both good and bad.

Let us start with an example. The figure 14 shows a piece of code containing three variations of a loop performing identical tasks: the first uses the LL, the second uses the classic STL and the third is a traditional for loop. When this example is timed using the gcc 2.95.2 on a Pentium 450 with basic -O3 optimizing on, the `copy_if` using the LL is slightly faster, approximately 5%, than the classic STL version. When the lambda code is timed against a hand coded for loop, it is slightly more than 2.4 times as slow. If the array size is increased to 16,000 elements to reduce the effect of the penalty of the extra function call, lambda is only slightly more than 1.5 times as slow.

[11] The LL may use inventive techniques, but we can't say the same about function names.

```
template<class Args, class CatchType1>
class lambda_functor_base<
        action<2, try_catch_action<
                    catch_list_action<CatchType1> > >,
        Args
      >
{
  Args args;
public:
  explicit lambda_functor_base(const Args &a) : args(a) {}

  template<RET,class A, class B, class C>
  RET call(A&a, B&b, C&c) const
  {
    try
    {
      return select(get<1>(args), a, b, c);
    }
    catch(CatchType1& e)
    {
      return
        return_or_throw<
          RET,
          typename tuple_element_type<2, Args>::type
        >::call(get<2>(args), a, b, e);
    }
  }
};
```

Figure 13: One of the `lambda_functor_base` specializations for exception handling.

```
int v[100] =  100 values , vv[100];
const int r1 = 30, r2 = 50;

// using lambda:
copy_if(v, &v[100], r1 <= free1 && free1 < r2);

// using classic STL:
struct in_range {
  int imin, imax;
  in_range(int x, int y) : imin(x), imax(y) {}
  bool operator()(int i) { return imin <= i && i < imax; }
};
copy_if(v, &v[100], in_range(r1, r2));

// hand coded for loop:
register int const * const end = &v[100];
register int *pp = vv;
register int const rr1 = r1;
register int const rr2 = r2;
for(int *p = v; p<end;++p)
 if (rr1 <= *p && *p < rr2) { *pp++ = *p; }
```

Figure 14: Three variations of the same loop.

This is the bad news. Now for the good news. When this same code is compiled with KCC, known to be one of the best optimizing C++ compilers, the lambda code is 5% faster than the hand written for loop!

When we add some more complexity, changing the comparison to:

```
copy_if(v, &v[100], r1 < (free1 + 3) && (free1 - 10) < r2);
```

using the gcc again, the lambda code becomes 15% slower than the classic STL version, but compared to the hand optimized version, is now slightly faster at 2.3 times as slow.[12]

The good news from all of this is that with a reasonable optimizing compiler, using simple lambda expressions you are no worse off than using classic STL. The other good news is that with a great optimizing compiler there is no penalty at all.

10.1 Current compiler limitations

The C++ Standard [C++98] suggests a template nesting level of 17 to help detect infinite recursion. Complex lambda templates can easily exceed this limit. Most compilers allow a greater number of nested templates, but may require you to increase this limit explicitly via a command line argument.

The current implementation of the Lambda Library uses member templates and partial specialization of templates. They are all standard features of the C++ language, but not all vendors support them at the time of writing this. We maintain a list of compilers that can compile the LL in our web page `http://lambda.cs.utu.fi`.

11 Relation to previous work

We've taken the operator overloading concept from the Expression Template library by Powell and Higley [PH00], the binding mechanism (see section 2.3) from the *Binder Library* (BL) by Järvi [Jär00]. We have combined, extended and generalized these ideas to create the Lambda Library.

Our implementation provides lambda functions which are both easier to use and more general than the expression templates in the ET library. The ET library used internally the STL template functors `plus`, `minus` etc. to handle the standard operators. These binary functors require that the arguments and return object be of the same type. The ET library also required that the user specify the type of the elements of the container for its placeholders and declare them before using them. These restrictions are lifted in the LL. Further, the binding mechanism is more general than the one described in [Jär00]. Specifically, even the target function can be left unbound (see section 2.3).

[12] We know it is a small performance gain but it is a gain.

Other related work includes the FACT [Fac00] and FC++ [MS00] libraries, both developed coevally with the LL. The basic idea behind FACT is quite similar to the LL, although the syntax of the lambda expressions is different. Compared to LL, FACT supports a smaller set of operators. FACT deliberately allows no side effects in lambda functions, which means that assignments and control constructs are not supported. FACT lambda functions are 'expression template aware' (see [SS00]), while LL lambda functions are not. Inspired by this feature in FACT, LL does provide a mechanism for user extensions to enable the use of other expression template libraries, like PETE [HCKS99], and Blitz++ [Bli99].

The FC++ is another library adding functional features to C++; it more or less embeds a functional sublanguage to C++. A unique feature of FC++ is the possibility to define variables for storing lambda functors. This is very convenient, even though the feature comes with some cost: the lambda functor becomes dynamically bound, and the return type and the argument types of the lambda functor must be defined explicitly by the client.

To our understanding, FACT and FC++ take the functional features in a 'pure' form. The LL implements lambda functions, but does some adjustments for a better fit to C++. Our foremost goal is to provide lambda functors which match perfectly with the STL style of programming, not to rigorously follow the functional programming paradigm.

FACT and FC++ introduce some very innovative template techniques and we urge the reader to take a look at both of these libraries.

12 Conclusion

With the Lambda Library we hope to provide a valuable set of tools for working with STL algorithms. The LL is quite a step forward from the ET [PH00] and BL [Jär00] libraries, not to mention the standard binders and functors.

The LL removes several restrictions and simplifies the use of 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 usage: 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.

Basically, all the above features are just subfeatures of a single language construct, the lambda abstraction. As it is not a built-in feature of C++, nor will be in the near future, we hope you will find comfort from the Lambda Library.

12.1 Future Work

We are planning to add some new features to the LL, partly inspired by some of the techniques used in the FACT [Fac00] and FC++ [MS00] libraries. We are also on the lookout for common idioms, which could benefit from becoming a lambda expression template and be incorporated into this library. In any case, the LL is still experimental, and thus subject to changes.

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 really benefit from a full compiler supported "typeof()" expression as it would eliminate a great deal of the complex return type deduction code. We looked at using a template library which emulates a true typeof operator [Gib00], but gcc 2.95.2 failed to compile it. When gcc releases the next major update we will revisit this.

There are two gray areas which we haven't really looked at yet, but we know we should, namely exception specifications in function prototypes and the volatile qualifier. We may also consider increasing the number of allowed arguments to tuples. For the most part, 10 arguments has been sufficient for our needs. Finally, there is work to do in porting the library to new compilers; we gladly welcome volunteers.

Acknowledgements

Gary's: Jaakko and I met via the boost list server. I had been working on expression templates and had not been satisfied with the implementation. At the same time I discovered Jaakko's bind library, Jaakko discovered me. The exchange of emails was the match to the tinder. We shared papers, and there has been no stopping us since. Jaakko wrote the core of the Lambda Library and recognized the need for a multilayered approach which we have extended to isolate each problem. I suggested a number of additions to the library which either I coded an initial attempt, some of which Jaakko fixed, and others he replaced with a more robust implementation. Some code is from both of us, but in general, Jaakko did most of it. I've learned a lot about using templates from Jaakko.

Jaakko's: Starting from modest improvements to the standard binders, we have steadily approached a proper, and efficient, lambda abstraction. It was truly interesting to discover that standard C++ had all the ingredients for it. All that was needed was template specialization, function overloading and some help from the compiler (e.g. heavy inlining) and a lot of help from Gary.

Gary's inexhaustible flow of new ideas has brought LL much further I could have ever imagined. I probably would have added control structures

at some point, but exception handling is not something you would first think of. Gary's constant help has been invaluable.

We are working with a 9-hour time difference, which has allowed us to code around the clock. Thus the sum of two individuals working together is greater than 2, it may even be 4. Also, we've worked together very hard for quite some time and have actually never met, so forgive us the abundance of mutual thanking.

Outside help: No project exists in a vacuum. We are indebted to those who have gone before us and helped us along the way. The return type promotion code of arithmetic types is modeled after that used in the Blitz++ library [Bli99]. Andrei Alexandrescu's clever code for testing type compatibility at compile time is used at some parts of the return type deduction system. Valentin Bonnard suggested a way to globally overload the `operator->*()` [Bon00]. Jeremy Siek has helped us with the performance testing, reviewed the code and ported it to KAI 3.4, and attempted a port to MSVC 6.3.; unfortunately it proved to be impossible. However, William Kempf is giving it another try, so *to prove* or *impossible* may prove to be subtle concepts.

Peter Higley has reviewed early versions of this paper, and made some good comments to add to its clarity. He also attempted a port to the Intel 4.0 compiler and it fails.

Bjarne Stroustrup's [Str00] comments have inspired us to divide an intermixed paper into two distinct parts. We would also like to thank the boost readership for their insightful comments, especially Dave Abrahams.

References

[Bli99] The Blitz++ library home page, 1999. `http://www.oonumerics.org/blitz`.

[Bon00] V. Bonnard. Correspondence via the Boost listserver, 2000.

[C++98] International Standard, Programming Languages – C++, ISO/IEC:14882, 1998.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming : Methods, Tools, and Applications*. Addison-Wesley, 2000.

[Fac00] The FACT! library home page, 2000. `http://www.fz-juelich.de/zam/FACT`.

[Gib00] B. Gibbons. A portable "typeof" operator. *C/C++ Users Journal*, November 2000.

[HCKS99] S. W. Haney, J. Crotinger, S. Karmesin, and S. Smith. Pete, the portable expression template engine. *Dr. Dobb's Journal*, pages 88–95, October 1999. `http://www.acl.lanl.gov/pete`.

[Jär99a] J. Järvi. Tuples and multiple return values in C++. Technical Report 249, Turku Centre for Computer Science, March 1999.

[Jär99b] J. Järvi. ML-style tuple assignment in standard C++ — extending the multiple return value formalism. Technical Report 267, Turku Centre for Computer Science, March 1999.

[Jär00] 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*, August 2000.

[Jär01] J. Järvi. Tuple types and multiple return values. *C/C++ Users Journal*, 2001. To appear.

[Mey96] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley, 1996.

[Mey99] S. Meyers. Implementing operator->* for smart pointers. *Dr. Dobb's Journal*, October 1999.

[MS00] B. McNamara and Y. Smaragdakis. Functional Programming in C++. In *Proceedings of The 2000 International Conference on Functional Programming (ICFP).* ACM, 2000. `http://www.cc.gatech.edu/~yannis/fc++`.

[PH00] G. Powell and P. Higley. Expression templates as a replacement for simple functors. *C++ Report*, 12(5), May 2000.

[SL94] 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`.

[SS00] J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 2000.

[STL00] The SGI Standard Template Library. `http://www.sgi.com/Technology/STL`, 2000.

[Str00] B. Stroustrup. Personal correspondence via email, 2000.

[Vel95] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**