The Lambda Library: Lambda Abstraction in C++

Jaakko Järvi
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14-18 A
FIN-20520 Turku, Finland
jaakko.jarvi@cs.utu.fi

Gary Powell
Sierra On-Line Ltd.
gwpowell@hotmail.com

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.

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 mainstream procedural or object-oriented languages (except for Eiffel, where *agents* [1], a recently added language feature provides means to define unnamed functions). In this paper we introduce 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) [2], now part of the C++ Standard Library [3]. This article introduces the features of the library rather than explains its inner workings. There is an accompanying technical report, which describes the features more thoroughly and goes over the implementation details [4].

1.1 Motivation

Typically STL algorithms operate on container elements via functions and function objects passed as arguments to the algorithms. 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, there are *binder* templates `bind1st` and `bind2nd` for creating unary function objects from binary function objects by *binding* one of the arguments to a constant value. Some STL implementations contain function composition operations as extensions to the standard [5].

The goal of all these tools is clear: to make it possible to specify unnamed functions in a call to an STL algorithm. However, this set of tools leaves much room for improvement. 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 [6, 7] for a more in-depth discussions 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, _1 * pi / 180.0));
```

The `operator-` replaces the call to `negate`, `bind` function replaces the `compose1` call, `ptr_fun` becomes unnecessary, along with `bind2nd`, and the call to `multiplies` is replaced with the `operator*`. The argument `_1` is a *placeholder* representing the parameter of the function object. At each iteration, it will be substituted by the element from the container of angles.

1.2 Relation to other work

The LL combines, extends and generalizes the functionalities of the *Expression Template library* (ET) by Powell and Higley [8] and the *Binder Library* by Järvi [6]. Other related work includes the FACT [9] and FC++ [10] libraries, both developed coevally with the LL. The basic idea behind the FACT lambda functions is quite similar to the LL counterparts, although the syntax is different. Compared to LL, FACT supports a smaller set of operators. FACT deliberately allows no side effects in lambda functions, which means that, e.g. various assignment operators are not supported. FACT lambda functions are 'expression template aware' (see [11]), while basic LL lambda functions are not. Interaction with other expression template libraries deserves some more research. Further, FACT provides other features in addition to lambda functions, such as lazy lists.

The FC++ is another library adding functional features to C++; it more or less embeds a functional sublanguage to C++. A notable feature of FC++ is the possibility to define variables for storing lambda functions. This is very convenient, even though the feature comes with some cost: the lambda function becomes dynamically bound, and the return type and the argument types of the lambda function must be defined explicitly by the client. The *Function Library* in *C++ Boost* [12] has a similar feature in a form that can be combined with other libraries, e.g. with the LL.

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 functions which match perfectly with the STL style of programming.

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(), _1 = 1);
```

In this example `_1 = 1` creates a lambda function which assigns the value `1` to every element in `v`; The variable `_1` is a placeholder, an empty slot, which will be filled with a value at each iteration. Regarding terminology, we call `_1 = 1` a *lambda expression.* A function object created by a lambda expression is a *lambda functor.*

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

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

Here we take the address of each element in `v` (with `&_1`) 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(),
         _1 = bind(foo, _1));
```

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

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

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

```
for_each(vp.begin(), vp.end(),
         cout << *_1 << endl);
```

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 LL counterpart of the above expression is written as:

```
_1 + _2
```

In this 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 `_1` and `_2`. The LL supports one more: `_3`. This means that lambda functors can take one, two or three arguments passed in by the STL algorithm; zero parameters is possible too. It would be straightforward to support higher arities, but 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 3.6, which introduces `_E`, another incarnation of `_3`).

[1] This doesn't hold for all expressions, see section 2.3

The placeholders are variables defined by the library. 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 when the lambda functor gets called. The LL provides typedefs for the placeholder types, so it is easy to define the placeholder names to your own liking.

Veldhuizen [13] was the first to introduce the concept of using placeholders in expression templates. The LL placeholders are somewhat different from the original ones. 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 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(), _1 = 1);
```

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

Since the type of the placeholder argument is left open, the return type of the lambda functor is not known either. 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.

2.3 Functions as lambda expressions

The use of a placeholder as one of the operands turns an operator invocation into a lambda expression implicitly. For ordinary function calls this is not the case; an explicit syntactic construct is needed. 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*.

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

2.3.1 Function pointers as targets

The target function can be a pointer or a reference to a non-member function (or a static 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, _1, _2, c)
bind(&foo, _1, _2, c)
bind(foo, _1, _1, _1)
bind(_1, a, b, c)
```

The first bind expression returns a binary lambda functor. The second bind expression has an equivalent functionality, it just uses a function pointer instead of a reference. The third bind expression 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 the case where the target function is 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, _1)(i); // error
  ...
void (*pf1)(int) = &foo;
bind(pf1, _1)(i); // ok
  ...
bind(static_cast<void(*)(int)>(&foo),
     _1)(i);        // ok as well
```

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 *adaptable* function objects in the STL. For example:

```
class A {
  ...
public:
 typedef B result_type;
 B operator()(X, Y, Z);
};
```

The above function object can be used as:

```
A a; X x; Y y; Z z; list<A> la; list<Z> lz;

for_each(lz.begin(), lz.end(),
         bind(a, x, y, _1));
for_each(la.begin(), la.end(),
         bind(_1, 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 the technical report [4]).

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. A bound target object 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, _1));

// pointer is ok:
find_if(ints.begin(), ints.end(),
        bind(&A::foo, ap, _1));
```

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

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

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

  A workaround for this situation is explained in section 2.5.

- As stated in section 2.2, the return type deduction system may not handle all user-defined operators. For example, the return type of all comparison operators is expected to be `bool`. If this is not true for some user-defined comparison operator, return type deduction fails. In such cases the deduction system can either be extended, or temporarily overridden by explicit type information (see the technical report [4]).

2.5 Delayed constants and variables

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

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

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

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

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

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

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

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

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

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

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

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

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

2.6 About bound arguments

Bound arguments of a lambda expression are stored in the lambda functor object. There are few alternative ways of doing this, and the choices have consequences on whether side effects to the bound arguments are allowed or not. Basically the lambda functors can store temporary copies of the arguments, or hold const or non-const references to them. The default is 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; (_1 + i)(i = 10);
```

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

As said, this is the default, and for some expressions it makes more sense to store the arguments as references and allow side effects to the arguments. As an example, consider the lambda expression `i += _1`. The obvious intention is that calls to the lambda functor affect the value of the variable `i`, rather than some temporary copy of it. The LL has this behavior: the left argument of the compound assignment operators (`+=`, `*=`, etc.) are stored as references to non-const. Further, to make the streaming operators (`<<` and `>>`) work, 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 to store temporary copies, except for the target object, which is stored as a reference.

For all cases, LL provides means for overriding the default storing mechanism. For example, any bound argument in a lambda expression can be wrapped with a function named `ref` to state that the lambda functor should store a reference to the argument. Regarding the preceding example, the lambda expression $\lambda x.x + i$ can be created with the aid of the `ref` function as `_1 + ref(i)`. For an in depth discussion about this issue, see [14].

3 Advanced features

Our goal has been to make the LL as complete as possible in the sense that any C++ expression could be turned into a lambda expression. This section describes how to 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.

3.1 Comma and logical operators

The LL overloads the comma operator for sequencing lambda expressions together, as did the ET library [8]. 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 << _1 << endl,
            clog << _1 << 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(),
         _1 < 0 && _1 = 0);
```

3.2 Other special operators

Since the ?: operator cannot be overloaded, we have created the function if_then_else_return to duplicate the ?: operator's functionality.

The function call operator is overloaded for placeholders only, since other lambda functions already define operator().

The pointer to member function (operator->*) is a special case in the sense, that it can refer either to a member object or a member function.

3.3 Control lambda expressions

The idea of providing lambda expression variants for control structures originates from the ET Library [8]. 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 provides switch_statement and do_once[2].

Control lambda expressions 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(_1 % 2 == 0, cout << _1));
```

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

[2]A special kind of do while construct, see [4].

```
for_loop(init, 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),
           _1[var(i)] += 6));
```

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

As stated in section 2.5, 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, _1[vi] += 6));
```

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

3.4 Switch

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

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

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

3.5 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, _1 = new_ptr<int>());
for_each(a, a+10, delete_ptr(_1));
```

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 (`x` and `y`), 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),
  constructor<pair<int, int> >(_1, _2));
```

3.6 Exception handling in lambda expressions

The LL allows you to create lambda functors that throw and catch exceptions. The form of a lambda expression for try catch blocks is as follows:

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

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

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

The figure 1. demonstrates the use of the LL exception handling tools. The first catch block is for

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

Figure 1: Throwing and handling exceptions.

handling exceptions of type `foo_ex`. Note the use of the `_1` placeholder in the lambda expression that defines the body of the handler.

The second handler catches exceptions from the standard library, writes an informative message to the `cout` stream and constructs and throws an exception of another type, `bar_ex`. An object of type `std::exception` carries a string explaining the cause of the exception. This explanation can be queried with the zero-argument `what` member function; `bind(&std::exception::what, _E)` is the lambda expression for creating the lambda functor for calling `what`. Note the use of `_E` as the argument. It is a special placeholder, which refers to the caught exception object within the handler body. `_E` is not a full-fledged placeholder, but rather a special case of `_3`. As a consequence, `_E` cannot be used outside of an exception handler lambda expression, and `_3` cannot be used inside of an exception handler lambda expression. Illegal use of placeholders is caught by the compiler. The last handler (`catch_all`) demonstrates rethrowing exceptions.

3.7 Nesting STL algorithms

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 3.3 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),
           _1[var(i)] += 6));
```

We could do better: the inner loop should really be another `for_each` invocation. However, `for_each` is a function template, and 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(_1, _1 + 10, _1 += 6));
```

Notice the reuse of `_1`. 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`.

4 About implementation

The LL implementation is based on *expression templates* [13]. The basic idea behind expression templates is to overload operators to create *expression objects* to represent the expression and its arguments instead of evaluating the operator instantly. As a result, the type of an expression object can describe a parse tree of the underlying expression. This expression object can be manipulated in various ways (also at compile time) prior to actually evaluating it, for example to yield better performance by preventing the creation of unnecessary temporary objects. 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.

As we said in section 2.2, LL placeholders do not carry information about the types of the open arguments. This leaves another task, return type deduction, to be performed by the expression template machinery. This is accomplished with a set of traits templates. As input, these templates take 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. The return type deduction templates are specialized with respect to the type of the target function (different templates for arithmetic operators, assignment operators, etc.). Each such specialization provides a mapping from the argument types to the return type. If the argument types are lambda functor instantiations as well, the compiler has to resort to the return type deduction system recursively to figure out the actual argument types.

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

The LL implementation is far from trivial. Especially, the return type deduction templates, as well as the argument substitution code are rather complicated. (The proposed language extension `typeof` would simplify the return type deduction code.) Further, allowing side effects via bound arguments, in other words passing and storing bound arguments as non-const references, requires quite some trickery. An interested reader should read the technical report [4] and take a look at the code. The library can be downloaded at `lambda.cs.utu.fi`.

5 About performance and use

In theory, all overhead of using STL algorithms and lambda functors compared to hand written loops can be optimized away, just as the overhead from standard STL function objects and binders. Depending on the compiler, this can also be true in practice. Our tests suggest that the LL does not introduce a loss of performance compared to STL function objects. Hence, with a reasonable optimizing compiler, using simple lambda expressions you are no worse off than using classic STL. Further, with a great optimizing compiler there is no penalty at all.

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

As another downside, compilation error messages that result from invalid lambda expressions can be very hard to comprehend. Even a very simple lambda functor has a type that spans several lines in an error message.

6 Conclusion

With the Lambda Library we hope to provide a valuable set of tools for working with STL algorithms. The LL removes several restrictions and simplifies the use of the STL in many ways. The users of the LL have a natural way to write simple functors cleanly. Teachers of STL algorithms can have students writing clear code quickly: it is easier to explain how to use the LL than the current alternative of `ptr_fun`, `mem_fun`, `bind1st`, etc. and it extends better to the more complex problems (cf. `bind3rdAnd4th`). Further, the LL introduces a set of entirely new possibilities for STL algorithm reuse: The LL makes it possible to loop through multiple nested containers. Exceptions can be thrown, caught and handled within the functor, and the looping in the STL algorithm can be continued. All the above features are built with standard C++ templates and do not change the design model of the language or require an additional preprocessing step.

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

References

[1] *Agents, iterators and introspection*, 2000. `http://www.eiffel.com` (information, papers).

[2] 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`.

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

[4] J. Järvi and G. Powell. The Lambda Library : Lambda abstraction in C++. Technical Report 378, Turku Centre for Computer Science, November 2000.

[5] The SGI Standard Template Library. `http://www.sgi.com/tech/stl`, 2000.

[6] J. Järvi. C++ function object binders made easy. In *Proceedings of the Generative and Component-Based Software Engineering'99*, volume 1799 of *Lecture Notes in Computer Science*, August 2000.

[7] V. Simonis. Adapters and binders - overcoming problems in the design and implementation of the C++-STL. *ACM SIGPLAN Notices*, January 2000.

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

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

[10] 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++`.

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

[12] C++ boost community. `http://www.boost.org`, 2000.

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

[14] J. Järvi and G. Powell. Side effects and partial function application in C++. In *Proceedings of the Multiparadigm Programming with OO Languages Workshop (MPOOL'01) at ECOOP 2001*, John von Neumann Institute of Computing series, 2001.