Boost.Lambda

Jaakko Järvi <jarvi at cs tamu edu>

Table of Contents

In a nutshell

The Boost Lambda Library (BLL in the sequel) is a C++ template library, which implements a form of *lambda abstractions* for C++. The term originates from functional programming and lambda calculus, where a lambda abstraction defines an unnamed function. The primary motivation for the BLL is to provide flexible and convenient means to define unnamed function objects for STL algorithms.

In explaining what the library is about, a line of code says more than a thousand words; the following line outputs the elements of some STL container a separated by spaces:

```
for_each(a.begin(), a.end(), std::cout << _1 << ' ');
```

The expression `std::cout << _1 << ' '` defines a unary function object. The variable `_1` is the parameter of this function, a *placeholder* for the actual argument. Within each iteration of `for_each`, the function is called with an element of `a` as the actual argument. This actual argument is substituted for the placeholder, and the "body" of the function is evaluated.

The essence of BLL is letting you define small unnamed function objects, such as the one above, directly on the call site of an STL algorithm.

Getting Started

Installing the library

The library consists of include files only, hence there is no installation procedure. The `boost` include directory must be on the include path. There are a number of include files that give different functionality:

- `lambda/lambda.hpp` defines lambda expressions for different C++ operators, see the section called "Operator expressions".
- `lambda/bind.hpp` defines `bind` functions for up to 9 arguments, see the section called "Bind expressions".
- `lambda/if.hpp` defines lambda function equivalents for if statements and the conditional operator, see the section called "Lambda expressions for control structures" (includes `lambda.hpp`).
- `lambda/loops.hpp` defines lambda function equivalent for looping constructs, see the section called "Lambda expressions for control structures".
- `lambda/switch.hpp` defines lambda function equivalent for the switch statement, see the section called "Lambda expressions for control structures".
- `lambda/construct.hpp` provides tools for writing lambda expressions with constructor, destructor, new and delete invocations, see the section called "Construction and destruction" (includes `lambda.hpp`).
- `lambda/casts.hpp` provides lambda versions of different casts, as well as `sizeof` and `typeid`, see the section called " Cast expressions ".
- `lambda/exceptions.hpp` gives tools for throwing and catching exceptions within lambda functions, the section called "Exceptions" (includes `lambda.hpp`).
- `lambda/algorithm.hpp` and `lambda/numeric.hpp` (cf. standard `algortihm` and `numeric` headers) allow nested STL algorithm invocations, see the section called "Nesting STL algorithm invocations".

Any other header files in the package are for internal use. Additionally, the library depends on two other Boost Libraries, the *Tuple* [tuple] and the *type_traits* [type_traits] libraries, and on the `boost/ref.hpp` header.

All definitions are placed in the namespace `boost::lambda` and its subnamespaces.

Conventions used in this document

In most code examples, we omit the namespace prefixes for names in the `std` and `boost::lambda` namespaces. Implicit using declarations

```
using namespace std;
using namespace boost::lambda;
```

are assumed to be in effect.

Introduction

Motivation

The Standard Template Library (STL) [STL94], now part of the C++ Standard Library [C++98], is a generic container and algorithm library. Typically STL algorithms operate on container elements via *function objects*. These function objects are passed as arguments to the algorithms.

Any C++ construct that can be called with the function call syntax is a function object. The STL contains predefined function objects for some common cases (such as `plus`, `less` and `not1`). As an example, one possible implementation for the standard `plus` template is:

```
template <class T>
struct plus : public binary_function<T, T, T> {
  T operator()(const T& i, const T& j) const {
    return i + j;
  }
};
```

The base class `binary_function<T, T, T>` contains typedefs for the argument and return types of the function object, which are needed to make the function object *adaptable*.

In addition to the basic function object classes, such as the one above, the STL contains *binder* templates for creating a unary function object from an adaptable binary function object by fixing one of the arguments to a constant value. For example, instead of having to explicitly write a function object class like:

```
class plus_1 {
  int _i;
public:
  plus_1(const int& i) : _i(i) {}
  int operator()(const int& j) { return _i + j; }
};
```

the equivalent functionality can be achieved with the `plus` template and one of the binder templates (`bind1st`). E.g., the following two expressions create function objects with identical functionalities; when invoked, both return the result of adding `1` to the argument of the function object:

```
plus_1(1)
bind1st(plus<int>(), 1)
```

The subexpression `plus<int>()` in the latter line is a binary function object which computes the sum of two integers, and `bind1st` invokes this function object partially binding the first argument to `1`. As an example of using the above function object, the following code adds `1` to each element of some container `a` and outputs the results into the standard output stream `cout`.

```
transform(a.begin(), a.end(), ostream_iterator<int>(cout),
          bind1st(plus<int>(), 1));
```

To make the binder templates more generally applicable, the STL contains *adaptors* for making pointers or references to functions, and pointers to member functions, adaptable. Finally, some STL implementations contain function composition operations as extensions to the standard [SGI02].

All these tools aim at one goal: to make it possible to specify *unnamed functions* in a call of an STL algorithm, in other words, to pass code fragments as an argument to a function. However, this goal is attained only partially. The simple example above shows that the definition of unnamed functions with the standard tools is cumbersome. Complex expressions involving functors, adaptors, binders and function composition operations tend to be difficult to comprehend. In addition to this, there are significant restrictions in applying the standard tools. E.g. 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.

The Boost Lambda Library provides solutions for the problems described above:

- Unnamed functions can be created easily with an intuitive syntax. The above example can be written as:

  ```
  transform(a.begin(), a.end(), ostream_iterator<int>(cout),
            1 + _1);
  ```

  or even more intuitively:

  ```
  for_each(a.begin(), a.end(), cout << (1 + _1));
  ```

- Most of the restrictions in argument binding are removed, arbitrary arguments of practically any C++ function can be bound.
- Separate function composition operations are not needed, as function composition is supported implicitly.

Introduction to lambda expressions

Lambda expression are common in functional programming languages. Their syntax varies between languages (and between different forms of lambda calculus), but the basic form of a lambda expressions is:

```
lambda x1 ... xn.e
```

A lambda expression defines an unnamed function and consists of:

- the parameters of this function: $x_1 \ldots x_n$.
- the expression e which computes the value of the function in terms of the parameters $x_1 \ldots x_n$.

A simple example of a lambda expression is

```
lambda x y.x+y
```

Applying the lambda function means substituting the formal parameters with the actual arguments:

```
(lambda x y.x+y) 2 3 = 2 + 3 = 5
```

In the C++ version of lambda expressions the `lambda` x_1 `...` x_n part is missing and the formal parameters have predefined names. In the current version of the library, there are three such predefined formal parameters, called *placeholders*: `_1`, `_2` and `_3`. They refer to the first, second and third argument of the function defined by the lambda expression. For example, the C++ version of the definition

```
lambda x y.x+y
```

is

```
_1 + _2
```

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

```
lambda x y.foo(x,y)
```

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

```
bind(foo, _1, _2)
```

We refer to this type of C++ lambda expressions as *bind expressions*.

A lambda expression defines a C++ function object, hence function application syntax is like calling any other function object, for instance: `(_1 + _2)(i, j)`.

Partial function application

A bind expression is in effect a *partial function application*. In partial function application, some of the arguments of a function are bound to fixed values. The result is another function, with possibly fewer arguments. When called with the unbound arguments, this new function invokes the original function with the merged argument list of bound and unbound arguments.

Terminology

A lambda expression defines a function. A C++ lambda expression concretely constructs a function object, *a functor*, when evaluated. We use the name *lambda functor* to refer to such a function object. Hence, in the terminology adopted here, the result of evaluating a lambda expression is a lambda functor.

Using the library

The purpose of this section is to introduce the basic functionality of the library. There are quite a lot of exceptions and special cases, but discussion of them is postponed until later sections.

Introductory Examples

In this section we give basic examples of using BLL lambda expressions in STL algorithm invocations. We start with some simple expressions and work up. First, we initialize the elements of a container, say, a `list`, to the value `1`:

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

The expression `_1 = 1` creates a lambda functor which assigns the value `1` to every element in `v`.[1]

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

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

The expression `&_1` creates a function object for getting the address of each element in `v`. The addresses get assigned to the corresponding elements in `vp`.

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

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

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

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

In this call to `sort`, we are sorting the elements by their contents in descending order.

Finally, the following `for_each` call outputs the sorted content of `vp` separated by line breaks:

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

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

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

the subexpression `cout << '\n'` is evaluated immediately and the effect is to output a single line break, followed by the elements of `vp`. The BLL provides functions `constant` and `var` to turn constants and, respectively, variables into lambda expressions, and can be used to prevent the immediate evaluation of subexpressions:

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

These functions are described more thoroughly in the section called “Delaying constants and variables”

[1] Strictly taken, the C++ standard defines `for_each` as a *non-modifying sequence operation*, and the function object passed to `for_each` should not modify its argument. The requirements for the arguments of `for_each` are unnecessary strict, since as long as the iterators are *mutable*, `for_each` accepts a function object that can have side-effects on their argument. Nevertheless, it is straightforward to provide another function template with the functionality of `std::for_each` but more fine-grained requirements for its arguments.

Parameter and return types of lambda functors

During the invocation of a lambda functor, the actual arguments are substituted for the placeholders. The placeholders do not dictate the type of these actual arguments. The basic rule is that a lambda function can be called with arguments of any types, as long as the lambda expression with substitutions performed is a valid C++ expression. As an example, the expression `_1 + _2` creates a binary lambda functor. It can be called with two objects of any types `A` and `B` for which `operator+(A,B)` is defined (and for which BLL knows the return type of the operator, see below).

C++ lacks a mechanism to query a type of an expression. However, this precise mechanism is crucial for the implementation of C++ lambda expressions. Consequently, BLL includes a somewhat complex type deduction system which uses a set of traits classes for deducing the resulting type of lambda functions. It handles expressions where the operands are of built-in types and many of the expressions with operands of standard library types. Many of the user defined types are covered as well, particularly if the user defined operators obey normal conventions in defining the return types.

There are, however, cases when the return type cannot be deduced. For example, suppose you have defined:

```
C operator+(A, B);
```

The following lambda function invocation fails, since the return type cannot be deduced:

```
A a; B b; (_1 + _2)(a, b);
```

There are two alternative solutions to this. The first is to extend the BLL type deduction system to cover your own types (see the section called “Extending return type deduction system”). The second is to use a special lambda expression (`ret`) which defines the return type in place (see the section called “Overriding the deduced return type”):

```
A a; B b; ret<C>(_1 + _2)(a, b);
```

For bind expressions, the return type can be defined as a template argument of the bind function as well:

```
bind<int>(foo, _1, _2);
```

About actual arguments to lambda functors

A general restriction for the actual arguments is that they cannot be non-const rvalues. For example:

```
int i = 1; int j = 2;
(_1 + _2)(i, j); // ok
(_1 + _2)(1, 2); // error (!)
```

This restriction is not as bad as it may look. Since the lambda functors are most often called inside STL-algorithms, the arguments originate from dereferencing iterators and the dereferencing operators seldom return rvalues. And for the cases where they do, there are workarounds discussed in the section called “Rvalues as actual arguments to lambda functors”.

Storing bound arguments in lambda functions

By default, temporary const copies of the bound arguments are stored in the lambda functor. This means that the value of a bound argument is fixed at the time of the creation of the lambda function and remains constant during the lifetime of the lambda function object. For example:

```
int i = 1;
(_1 = 2, _1 + i)(i);
```

The comma operator is overloaded to combine lambda expressions into a sequence; the resulting unary lambda functor first assigns 2 to its argument, then adds the value of `i` to it. The value of the expression in the last line is 3, not 4. In other words, the lambda expression that is created is `lambda x.(x = 2, x + 1)` rather than `lambda x.(x = 2, x + i)`.

As said, this is the default behavior for which there are exceptions. The exact rules are as follows:

- The programmer can control the storing mechanism with `ref` and `cref` wrappers [ref]. Wrapping an argument with `ref`, or `cref`, instructs the library to store the argument as a reference, or as a reference to const respectively. For example, if we rewrite the previous example and wrap the variable `i` with `ref`, we are creating the lambda expression `lambda x.(x = 2, x + i)` and the value of the expression in the last line will be 4:

  ```
  i = 1;
  (_1 = 2, _1 + ref(i))(i);
  ```

  Note that `ref` and `cref` are different from `var` and `constant`. While the latter ones create lambda functors, the former do not. For example:

  ```
  int i;
  var(i) = 1; // ok
  ref(i) = 1; // not ok, ref(i) is not a lambda functor
  ```

  The functions `ref` and `cref` mostly exist for historical reasons, and `ref` can always be replaced with `var`, and `cref` with `constant_ref`. See the section called “Delaying constants and variables” for details. The `ref` and `cref` functions are general purpose utility functions in Boost, and hence defined directly in the `boost` namespace.

- Array types cannot be copied, they are thus stored as const reference by default.

- For some expressions it makes more sense to store the arguments as references. For example, the obvious intention of the lambda expression `i += _1` is that calls to the lambda functor affect the value of the variable `i`, rather than some temporary copy of it. As another example, the streaming operators take their leftmost argument as non-const references. The exact rules are:

  - The left argument of compound assignment operators (+=, *=, etc.) are stored as references to non-const.

  - If the left argument of << or >> operator is derived from an instantiation of `basic_ostream` or respectively from `basic_istream`, the argument is stored as a reference to non-const. For all other types, the argument is stored as a copy.

  - In pointer arithmetic expressions, non-const array types are stored as non-const references. This is to prevent pointer arithmetic making non-const arrays const.

Lambda expressions in details

This section describes different categories of lambda expressions in details. We devote a separate section for each of the possible forms of a lambda expression.

Placeholders

The BLL defines three placeholder types: `placeholder1_type`, `placeholder2_type` and `placeholder3_type`. BLL has a predefined placeholder variable for each placeholder type: `_1`, `_2` and `_3`. However, the user is not forced to use these placeholders. It is easy to define placeholders with alternative names. This is done by defining new variables of placeholder types. For example:

```
boost::lambda::placeholder1_type X;
boost::lambda::placeholder2_type Y;
boost::lambda::placeholder3_type Z;
```

With these variables defined, `X += Y * Z` is equivalent to `_1 += _2 * _3`.

The use of placeholders in the lambda expression determines whether the resulting function is nullary, unary, binary or 3-ary. The highest placeholder index is decisive. For example:

```
_1 + 5              // unary
_1 * _1 + _1        // unary
_1 + _2             // binary
bind(f, _1, _2, _3) // 3-ary
_3 + 10             // 3-ary
```

Note that the last line creates a 3-ary function, which adds `10` to its *third* argument. The first two arguments are discarded. Furthermore, lambda functors only have a minimum arity. One can always provide more arguments (up the number of supported placeholders) that is really needed. The remaining arguments are just discarded. For example:

```
int i, j, k;
_1(i, j, k)        // returns i, discards j and k
(_2 + _2)(i, j, k) // returns j+j, discards i and k
```

See the section called " Lambda functor arity " for the design rationale behind this functionality.

In addition to these three placeholder types, there is also a fourth placeholder type `placeholderE_type`. The use of this placeholder is defined in the section called "Exceptions" describing exception handling in lambda expressions.

When an actual argument is supplied for a placeholder, the parameter passing mode is always by reference. This means that any side-effects to the placeholder are reflected to the actual argument. For example:

```
int i = 1;
(_1 += 2)(i);         // i is now 3
(++_1, cout << _1)(i) // i is now 4, outputs 4
```

Operator expressions

The basic rule is that any C++ operator invocation with at least one argument being a lambda expression is itself a lambda expression. Almost all overloadable operators are supported. For example, the following is a valid lambda expression:

```
cout << _1, _2[_3] = _1 && false
```

However, there are some restrictions that originate from the C++ operator overloading rules, and some special cases.

Operators that cannot be overloaded

Some operators cannot be overloaded at all (`::`, `.`, `.*`). For some operators, the requirements on return types prevent them to be overloaded to create lambda functors. These operators are `->.`, `->`, `new`, `new[]`, `delete`, `delete[]` and `?:` (the conditional operator).

Assignment and subscript operators

These operators must be implemented as class members. Consequently, the left operand must be a lambda expression. For example:

```
int i;
_1 = i;      // ok
i = _1;      // not ok. i is not a lambda expression
```

There is a simple solution around this limitation, described in the section called “Delaying constants and variables”. In short, the left hand argument can be explicitly turned into a lambda functor by wrapping it with a special `var` function:

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

Logical operators

Logical operators obey the short-circuiting evaluation rules. For example, in the following code, `i` is never incremented:

```
bool flag = true; int i = 0;
(_1 || ++_2)(flag, i);
```

Comma operator

Comma operator is the “statement separator” in lambda expressions. Since comma is also the separator between arguments in a function call, extra parenthesis are sometimes needed:

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

Without the extra parenthesis around `++_1, cout << _1`, the code would be interpreted as an attempt to call `for_each` with four arguments.

The lambda functor created by the comma operator adheres to the C++ rule of always evaluating the left operand before the right one. In the above example, each element of `a` is first incremented, then written to the stream.

Function call operator

The function call operators have the effect of evaluating the lambda functor. Calls with too few arguments lead to a compile time error.

Member pointer operator

The member pointer operator `operator->*` can be overloaded freely. Hence, for user defined types, member pointer operator is no special case. The built-in meaning, however, is a somewhat more complicated case. The built-in member pointer operator is applied if the left argument is a pointer to an object of some class `A`, and the right hand argument is a pointer to a member of `A`, or a pointer to a member of a class from which `A` derives. We must separate two cases:

- The right hand argument is a pointer to a data member. In this case the lambda functor simply performs the argument substitution and calls the built-in member pointer operator, which returns a reference to the member pointed to. For example:

  ```
  struct A { int d; };
  A* a = new A();
    ...
  (a ->* &A::d);     // returns a reference to a->d
  (_1 ->* &A::d)(a); // likewise
  ```

- The right hand argument is a pointer to a member function. For a built-in call like this, the result is kind of a delayed member function call. Such an expression must be followed by a function argument list, with which the delayed member function call is performed. For example:

```
struct B { int foo(int); };
B* b = new B();
  ...
(b ->* &B::foo)         // returns a delayed call to b->foo
                        // a function argument list must follow
(b ->* &B::foo)(1)      // ok, calls b->foo(1)

(_1 ->* &B::foo)(b);    // returns a delayed call to b->foo,
                        // no effect as such
(_1 ->* &B::foo)(b)(1); // calls b->foo(1)
```

Bind expressions

Bind expressions can have two forms:

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

A bind expression delays the call of a function. If this *target function* is *n*-ary, then the *`bind-argument-list`* must contain *n* arguments as well. In the current version of the BLL, 0 <= n <= 9 must hold. For member functions, the number of arguments must be at most 8, as the object argument takes one argument position. Basically, the *`bind-argument-list`* must be a valid argument list for the target function, except that any argument can be replaced with a placeholder, or more generally, with a lambda expression. Note that also the target function can be a lambda expression. The result of a bind expression is either a nullary, unary, binary or 3-ary function object depending on the use of placeholders in the *`bind-argument-list`* (see the section called “Placeholders”).

The return type of the lambda functor created by the bind expression can be given as an explicitly specified template parameter, as in the following example:

```
bind<RET>(target-function, bind-argument-list)
```

This is only necessary if the return type of the target function cannot be deduced.

The following sections describe the different types of bind expressions.

Function pointers or references as targets

The target function can be a pointer or a reference to a function and it can be either bound or unbound. For example:

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

The return type deduction always succeeds with this type of bind expressions.

Note, that in C++ it is possible to take the address of an overloaded function only if the address is assigned to, or used as an initializer of, a variable, the type of which solves the amibiguity, or if an explicit cast expression is used. This means that overloaded functions cannot be used in bind expressions directly, e.g.:

```
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
```

Member functions as targets

The syntax for using pointers to member function in bind expression is:

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

The object argument can be a reference or pointer to the object, the BLL supports both cases with a uniform interface:

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

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

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

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

```
class A {
  int i; mutable int j;
public:

  A(int ii, int jj) : i(ii), j(jj) {};
  void set_i(int x) { i = x; };
  void set_j(int x) const { j = x; };
};
```

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

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

Even though a const copy of the object argument is stored, the original object `a` is still modified. This is since the object argument is a pointer, and the pointer is copied, not the object it points to. When we use a reference, the behaviour is different:

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

To prevent the copying from taking place, one can use the `ref` or `cref` wrappers (`var` and `constant_ref` would do as well):

```
bind(&A::set_i, ref(a), _1)(k); // a.j == 1
bind(&A::set_j, cref(a), _1)(k); // a.j == 1
```

Note that the preceding discussion is relevant only for bound arguments. If the object argument is unbound, the parameter passing mode is always by reference. Hence, the argument `a` is not copied in the calls to the two lambda functors below:

```
A a(0,0);
bind(&A::set_i, _1, 1)(a); // a.i == 1
bind(&A::set_j, _1, 1)(a); // a.j == 1
```

Member variables as targets

A pointer to a member variable is not really a function, but the first argument to the `bind` function can nevertheless be a pointer to a member variable. Invoking such a bind expression returns a reference to the data member. For example:

```
struct A { int data; };
A a;
bind(&A::data, _1)(a) = 1;     // a.data == 1
```

The cv-qualifiers of the object whose member is accessed are respected. For example, the following tries to write into a const location:

```
const A ca = a;
bind(&A::data, _1)(ca) = 1;     // error
```

Function objects as targets

Function objects, that is, class objects which have the function call operator defined, can be used as target functions. In general, BLL cannot deduce the return type of an arbitrary function object. However, there are two methods for giving BLL this capability for a certain function object class.

The result_type typedef

The BLL supports the standard library convention of declaring the return type of a function object with a member typedef named `result_type` in the function object class. Here is a simple example:

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

If a function object does not define a `result_type` typedef, the method described below (`sig` template) is attempted to resolve the return type of the function object. If a function object defines both `result_type` and `sig`, `result_type` takes precedence.

The sig template

Another mechanism that make BLL aware of the return type(s) of a function object is defining member template struct `sig<Args>` with a typedef `type` that specifies the return type. Here is a simple example:

```
struct A {
  template <class Args> struct sig { typedef B type; }
  B operator()(X, Y, Z);
};
```

The template argument `Args` is a `tuple` (or more precisely a `cons` list) type [tuple], where the first element is the function object type itself, and the remaining elements are the types of the arguments, with which the function object is being called. This may seem overly complex compared to defining the `result_type` typedef. Howver, there are two significant restrictions with using just a simple typedef to express the return type:

1. If the function object defines several function call operators, there is no way to specify different result types for them.
2. If the function call operator is a template, the result type may depend on the template parameters. Hence, the typedef ought to be a template too, which the C++ language does not support.

The following code shows an example, where the return type depends on the type of one of the arguments, and how that dependency can be expressed with the `sig` template:

```
struct A {

  // the return type equals the third argument type:
  template<class T1, class T2, class T3>
  T3 operator()(const T1& t1, const T2& t2, const T3& t3) const;

  template <class Args>
  class sig {
    // get the third argument type (4th element)
    typedef typename
      boost::tuples::element<3, Args>::type T3;
  public:
    typedef typename
      boost::remove_cv<T3>::type type;
  };
};
```

The elements of the `Args` tuple are always non-reference types. Moreover, the element types can have a const or volatile qualifier (jointly referred to as *cv-qualifiers*), or both. This is since the cv-qualifiers in the arguments can affect the return type. The reason for including the potentially cv-qualified function object type itself into the `Args` tuple, is that the function object class can contain both const and non-const (or volatile, even const volatile) function call operators, and they can each have a different return type.

The `sig` template can be seen as a *meta-function* that maps the argument type tuple to the result type of the call made with arguments of the types in the tuple. As the example above demonstrates, the template can end up being somewhat complex. Typical tasks to be performed are the extraction of the relevant types from the tuple, removing cv-qualifiers etc. See the Boost type_traits [type_traits]

and Tuple [type_traits] libraries for tools that can aid in these tasks. The `sig` templates are a refined version of a similar mechanism first introduced in the FC++ library [fc++].

Overriding the deduced return type

The return type deduction system may not be able to deduce the return types of some user defined operators or bind expressions with class objects. A special lambda expression type is provided for stating the return type explicitly and overriding the deduction system. To state that the return type of the lambda functor defined by the lambda expression `e` is `T`, you can write:

```
ret<T>(e);
```

The effect is that the return type deduction is not performed for the lambda expression `e` at all, but instead, `T` is used as the return type. Obviously `T` cannot be an arbitrary type, the true result of the lambda functor must be implicitly convertible to `T`. For example:

```
A a; B b;
C operator+(A, B);
int operator*(A, B);
  ...
ret<D>(_1 + _2)(a, b);     // error (C cannot be converted to D)
ret<C>(_1 + _2)(a, b);     // ok
ret<float>(_1 * _2)(a, b); // ok (int can be converted to float)
  ...
struct X {
  Y operator(int)();
};
  ...
X x; int i;
bind(x, _1)(i);            // error, return type cannot be deduced
ret<Y>(bind(x, _1))(i);    // ok
```

For bind expressions, there is a short-hand notation that can be used instead of `ret`. The last line could alternatively be written as:

```
bind<Z>(x, _1)(i);
```

This feature is modeled after the Boost Bind library [bind].

Note that within nested lambda expressions, the `ret` must be used at each subexpression where the deduction would otherwise fail. For example:

```
A a; B b;
C operator+(A, B); D operator-(C);
  ...
ret<D>( - (_1 + _2))(a, b); // error
ret<D>( - ret<C>(_1 + _2))(a, b); // ok
```

If you find yourself using `ret` repeatedly with the same types, it is worth while extending the return type deduction (see the section called “Extending return type deduction system”).

Nullary lambda functors and ret

As stated above, the effect of `ret` is to prevent the return type deduction to be performed. However, there is an exception. Due to the way the C++ template instantiation works, the compiler is always forced to instantiate the return type deduction templates for zero-argument lambda functors. This introduces a slight problem with `ret`, best described with an example:

```
struct F { int operator()(int i) const; };
F f;
  ...
bind(f, _1);           // fails, cannot deduce the return type
ret<int>(bind(f, _1)); // ok
  ...
bind(f, 1);            // fails, cannot deduce the return type
ret<int>(bind(f, 1));  // fails as well!
```

The BLL cannot deduce the return types of the above bind calls, as `F` does not define the typedef `result_type`. One would expect `ret` to fix this, but for the nullary lambda functor that results from a bind expression (last line above) this does not work. The return type deduction templates are instantiated, even though it would not be necessary and the result is a compilation error.

The solution to this is not to use the `ret` function, but rather define the return type as an explicitly specified template parameter in the `bind` call:

```
bind<int>(f, 1);       // ok
```

The lambda functors created with `ret<`*`T`*`>(bind(`*`arg-list`*`))` and `bind<`*`T`*`>(`*`arg-list`*`)` have the exact same functionality — apart from the fact that for some nullary lambda functors the former does not work while the latter does.

Delaying constants and variables

The unary functions `constant`, `constant_ref` and `var` turn their argument into a lambda functor, that implements an identity mapping. The former two are for constants, the latter for variables. The use of these *delayed* constants and variables is sometimes necessary due to the lack of explicit syntax for lambda expressions. For example:

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

The first line outputs the elements of `a` separated by spaces, while the second line outputs a space followed by the elements of `a` without any separators. The reason for this is that neither of the operands of `cout << ' '` is a lambda expression, hence `cout << ' '` is evaluated immediately. To delay the evaluation of `cout << ' '`, one of the operands must be explicitly marked as a lambda expression. This is accomplished with the `constant` function:

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

The call `constant(' ')` creates a nullary lambda functor which stores the character constant `' '` and returns a reference to it when invoked. The function `constant_ref` is similar, except that it stores a constant reference to its argument. The `constant` and `consant_ref` are only needed when the operator call has side effects, like in the above example.

Sometimes we need to delay the evaluation of a variable. Suppose we wanted to output the elements of a container in a numbered list:

```
int index = 0;
for_each(a.begin(), a.end(), cout << ++index << ':' << _1 << '\n');
for_each(a.begin(), a.end(), cout << ++var(index) << ':' << _1 << '\n');
```

The first `for_each` invocation does not do what we want; `index` is incremented only once, and its value is written into the output stream only once. By using `var` to make `index` a lambda expression, we get the desired effect.

In sum, `var(x)` creates a nullary lambda functor, which stores a reference to the variable `x`. When the lambda functor is invoked, a reference to `x` is returned.

Naming delayed constants and variables

It is possible to predefine and name a delayed variable or constant outside a lambda expression. The templates `var_type`, `constant_type` and `constant_ref_type` serve for this purpose. They are used as:

```
var_type<T>::type delayed_i(var(i));
constant_type<T>::type delayed_c(constant(c));
```

The first line defines the variable `delayed_i` which is a delayed version of the variable `i` of type `T`. Analogously, the second line defines the constant `delayed_c` as a delayed version of the constant `c`. For example:

```
int i = 0; int j;
for_each(a.begin(), a.end(), (var(j) = _1, _1 = var(i), var(i) = var(j)));
```

is equivalent to:

```
int i = 0; int j;
var_type<int>::type vi(var(i)), vj(var(j));
for_each(a.begin(), a.end(), (vj = _1, _1 = vi, vi = vj));
```

Here is an example of naming a delayed constant:

```
constant_type<char>::type space(constant(' '));
for_each(a.begin(),a.end(), cout << space << _1);
```

About assignment and subscript operators

As described in the section called “Assignment and subscript operators”, assignment and subscripting operators are always defined as member functions. This means, that for expressions of the form `x = y` or `x[y]` to be interpreted as lambda expressions, the left-hand operand `x` must be a lambda expression. Consequently, it is sometimes necessary to use `var` for this purpose. We repeat the example from the section called “Assignment and subscript operators”:

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

Note that the compound assignment operators `+=`, `-=` etc. can be defined as non-member functions, and thus they are interpreted as lambda expressions even if only the right-hand operand is a lambda expression. Nevertheless, it is perfectly ok to delay the left operand explicitly. For example, `i += _1` is equivalent to `var(i) += _1`.

Lambda expressions for control structures

BLL defines several functions to create lambda functors that represent control structures. They all take lambda functors as parameters and return `void`. To start with an 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));
```

The BLL supports the following function templates for control structures:

```
if_then(condition, then_part)
if_then_else(condition, then_part, else_part)
if_then_else_return(condition, then_part, else_part)
while_loop(condition, body)
while_loop(condition) // no body case
do_while_loop(condition, body)
do_while_loop(condition) // no body case
for_loop(init, condition, increment, body)
for_loop(init, condition, increment) // no body case
switch_statement(...)
```

The return types of all control construct lambda functor is `void`, except for `if_then_else_return`, which wraps a call to the conditional operator

```
condition ? then_part : else_part
```

The return type rules for this operator are somewhat complex. Basically, if the branches have the same type, this type is the return type. If the type of the branches differ, one branch, say of type `A`, must be convertible to the other branch, say of type `B`. In this situation, the result type is `B`. Further, if the common type is an lvalue, the return type will be an lvalue too.

Delayed variables tend to be commonplace in control structure lambda expressions. For instance, here we use the `var` function to turn the arguments of `for_loop` into lambda expressions. The effect of the code is to add 1 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)] += 1));
```

The BLL supports an alternative syntax for control expressions, suggested by Joel de Guzmann. By overloading the `operator[]` we can get a closer resemblance with the built-in control structures:

```
if_(condition)[then_part]
if_(condition)[then_part].else_[else_part]
while_(condition)[body]
do_[body].while_(condition)
for_(init, condition, increment)[body]
```

For example, using this syntax the `if_then` example above can be written as:

```
for_each(a.begin(), a.end(),
         if_(_1 % 2 == 0)[ cout << _1 ])
```

As more experience is gained, we may end up deprecating one or the other of these syntaces.

Switch statement

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

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

The *`condition`* argument must be a lambda expression that creates a lambda functor with an integral return type. The different cases are created with the `case_statement` functions, and the optional default case with the `default_statement` function. The case labels are given as explicitly specified template arguments to `case_statement` functions and `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;
```

The `switch_statement` function is specialized for up to 9 case statements.

As a concrete example, the following code iterates over some container `v` and ouptuts “zero” for each `0`, “one” for each `1`, and “other: *n*” for any other value *n*. Note that another lambda expression is sequenced after the `switch_statement` to output a line break after each element:

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

Exceptions

The BLL provides lambda functors that throw and catch exceptions. 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.

Lambda expressions for handling exceptions are somewhat more complex. The general 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 where the explicitly specified template argument defines the type of the exception to catch. The lambda expression within the `catch_exception` defines the actions to take if the exception is caught. Note that the resulting exception handlers catch the exceptions as references, i.e., `catch_exception<T>(...)` results in the catch block:

```
catch(T& e) { ... }
```

The last catch block can alternatively be a call to `catch_exception<type>` or to `catch_all`, which is the lambda expression equivalent to `catch(...)`.

The Example 1, “Throwing and handling exceptions in lambda expressions.” demonstrates the use of the BLL exception handling tools. The first handler catches exceptions of type `foo_exception`. Note the use of `_1` placeholder in the body of the handler.

The second handler shows how to throw exceptions, and demonstrates the use of the *exception placeholder* `_e`. It is a special placeholder, which refers to the caught exception object within the handler body. Here we are handling an exception of type `std::exception`, which carries a string explaining the cause of the exception. This explanation can be queried with the zero-argument member function `what`. The expression `bind(&std::exception::what, _e)` creates the lambda function for making that call. Note that `_e` cannot be used outside of an exception handler lambda expression. The last line of the second handler constructs a new exception object and throws that with `throw exception`. Constructing and destructing objects within lambda expressions is explained in the section called “Construction and destruction”

Finally, the third handler (`catch_all`) demonstrates rethrowing exceptions.

Example 1. Throwing and handling exceptions in lambda expressions.

```
for_each(
  a.begin(), a.end(),
  try_catch(
    bind(foo, _1),                  // foo may throw
    catch_exception<foo_exception>(
      cout << constant("Caught foo_exception: ")
           << "foo was called with argument = " << _1
    ),
    catch_exception<std::exception>(
      cout << constant("Caught std::exception: ")
           << bind(&std::exception::what, _e),
      throw_exception(bind(constructor<bar_exception>(), _1)))
    ),
    catch_all(
      (cout << constant("Unknown"), rethrow())
    )
  )
);
```

Construction and destruction

Operators `new` and `delete` can be overloaded, but their return types are fixed. Particularly, the return types cannot be lambda functors, which prevents them to be overloaded for lambda expressions. It is not possible to take the address of a constructor, hence constructors cannot be used as target functions in bind expressions. The same is true for destructors. As a way around these constraints, BLL defines wrapper classes for `new` and `delete` calls, as well as for constructors and destructors. Instances of these classes are function objects, that can be used as target functions of bind expressions. For example:

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

The `new_ptr<int>()` expression creates a function object that calls `new int()` when invoked, and wrapping that inside `bind` makes it a lambda functor. In the same way, the expression `delete_ptr()` creates a function object that invokes `delete` on its argument. Note that `new_ptr<`*`T`*`>()` can take arguments as well. They are passed directly to the constructor invocation and thus allow calls to constructors which take arguments.

As an example of constructor calls in lambda expressions, the following code 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),
          bind(constructor<pair<int, int> >(), _1, _2));
```

Table 1, “Construction and destruction related function objects.” lists all the function objects related to creating and destroying objects, showing the expression to create and call the function object, and the effect of evaluating that expression.

Table 1. Construction and destruction related function objects.

Function object call	Wrapped expression
`constructor<T>()(arg_list)`	T(*arg_list*)
`destructor()(a)`	`a.~A()`, where `a` is of type `A`
`destructor()(pa)`	`pa->~A()`, where `pa` is of type `A*`
`new_ptr<T>()(arg_list)`	`new T(arg_list)`
`new_array<T>()(sz)`	`new T[sz]`
`delete_ptr()(p)`	`delete p`
`delete_array()(p)`	`delete p[]`

Special lambda expressions

Preventing argument substitution

When a lambda functor is called, the default behavior is to substitute the actual arguments for the placeholders within all subexpressions. This section describes the tools to prevent the substitution and evaluation of a subexpression, and explains when these tools should be used.

The arguments to a bind expression can be arbitrary lambda expressions, e.g., other bind expressions. For example:

```
int foo(int); int bar(int);
...
int i;
bind(foo, bind(bar, _1))(i);
```

The last line makes the call `foo(bar(i));` Note that the first argument in a bind expression, the target function, is no exception, and can thus be a bind expression too. The innermost lambda functor just has to return something that can be used as a target function: another lambda functor, function pointer, pointer to member function etc. For example, in the following code the innermost lambda functor makes a selection between two functions, and returns a pointer to one of them:

```
int add(int a, int b) { return a+b; }
int mul(int a, int b) { return a*b; }

int(*)(int, int)  add_or_mul(bool x) {
  return x ? add : mul;
}

bool condition; int i; int j;
...
bind(bind(&add_or_mul, _1), _2, _3)(condition, i, j);
```

Unlambda

A nested bind expression may occur inadvertently, if the target function is a variable with a type that depends on a template parameter. Typically the target function could be a formal parameter of a function template. In such a case, the programmer may not know whether the target function is a lambda functor or not.

Consider the following function template:

```
template<class F>
int nested(const F& f) {
  int x;
  ...
  bind(f, _1)(x);
  ...
}
```

Somewhere inside the function the formal parameter `f` is used as a target function in a bind expression. In order for this `bind` call to be valid, `f` must be a unary function. Suppose the following two calls to `nested` are made:

```
int foo(int);
int bar(int, int);
nested(&foo);
nested(bind(bar, 1, _1));
```

Both are unary functions, or function objects, with appropriate argument and return types, but the latter will not compile. In the latter call, the bind expression inside `nested` will become:

```
bind(bind(bar, 1, _1), _1)
```

When this is invoked with `x`, after substituitions we end up trying to call

```
bar(1, x)(x)
```

which is an error. The call to `bar` returns int, not a unary function or function object.

In the example above, the intent of the bind expression in the `nested` function is to treat `f` as an ordinary function object, instead of a lambda functor. The BLL provides the function template `unlambda` to express this: a lambda functor wrapped inside `unlambda` is not a lambda functor anymore, and does not take part into the argument substitution process. Note that for all other argument types `unlambda` is an identity operation, except for making non-const objects const.

Using `unlambda`, the `nested` function is written as:

```
template<class F>
int nested(const F& f) {
  int x;
  ...
  bind(unlambda(f), _1)(x);
  ...
}
```

Protect

The `protect` function is related to unlambda. It is also used to prevent the argument substitution taking place, but whereas `unlambda` turns a lambda functor into an ordinary function object for good, `protect` does this temporarily, for just one evaluation round. For example:

```
int x = 1, y = 10;
(_1 + protect(_1 + 2))(x)(y);
```

The first call substitutes `x` for the leftmost `_1`, and results in another lambda functor `x + (_1 + 2)`, which after the call with `y` becomes `x + (y + 2)`, and thus finally 13.

Primary motivation for including `protect` into the library, was to allow nested STL algorithm invocations (the section called "Nesting STL algorithm invocations").

Rvalues as actual arguments to lambda functors

Actual arguments to the lambda functors cannot be non-const rvalues. This is due to a deliberate design decision: either we have this restriction, or there can be no side-effects to the actual arguments. There are ways around this limitation. We repeat the example from section the section called "About actual arguments to lambda functors" and list the different solutions:

```
int i = 1; int j = 2;
(_1 + _2)(i, j); // ok
(_1 + _2)(1, 2); // error (!)
```

1. If the rvalue is of a class type, the return type of the function that creates the rvalue should be defined as const. Due to an unfortunate language restriction this does not work for built-in types, as built-in rvalues cannot be const qualified.

2. If the lambda function call is accessible, the `make_const` function can be used to *constify* the rvalue. E.g.:

   ```
   (_1 + _2)(make_const(1), make_const(2)); // ok
   ```

   Commonly the lambda function call site is inside a standard algorithm function template, preventing this solution to be used.

3. If neither of the above is possible, the lambda expression can be wrapped in a `const_parameters` function. It creates another type of lambda functor, which takes its arguments as const references. For example:

   ```
   const_parameters(_1 + _2)(1, 2); // ok
   ```

   Note that `const_parameters` makes all arguments const. Hence, in the case were one of the arguments is a non-const rvalue, and another argument needs to be passed as a non-const reference, this approach cannot be used.

4. If none of the above is possible, there is still one solution, which unfortunately can break const correctness. The solution is yet another lambda functor wrapper, which we have named `break_const` to alert the user of the potential dangers of this function. The `break_const` function creates a lambda functor that takes its arguments as const, and casts away constness prior to the call to the original wrapped lambda functor. For example:

   ```
   int i;
   ...
   (_1 += _2)(i, 2);                 // error, 2 is a non-const rvalue
   const_parameters(_1 += _2)(i, 2); // error, i becomes const
   break_const(_1 += _2)(i, 2);      // ok, but dangerous
   ```

   Note, that the results of `break_const` or `const_parameters` are not lambda functors, so they cannot be used as subexpressions of lambda expressions. For instance:

```
break_const(_1 + _2) + _3; // fails.
const_parameters(_1 + _2) + _3; // fails.
```

However, this kind of code should never be necessary, since calls to sub lambda functors are made inside the BLL, and are not affected by the non-const rvalue problem.

Casts, sizeof and typeid

Cast expressions

The BLL defines its counterparts for the four cast expressions `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`. The BLL versions of the cast expressions have the prefix `ll_`. The type to cast to is given as an explicitly specified template argument, and the sole argument is the expression from which to perform the cast. If the argument is a lambda functor, the lambda functor is evaluated first. For example, the following code uses `ll_dynamic_cast` to count the number of `derived` instances in the container `a`:

```
class base {};
class derived : public base {};

vector<base*> a;
...
int count = 0;
for_each(a.begin(), a.end(),
         if_then(ll_dynamic_cast<derived*>(_1), ++var(count)));
```

Sizeof and typeid

The BLL counterparts for these expressions are named `ll_sizeof` and `ll_typeid`. Both take one argument, which can be a lambda expression. The lambda functor created wraps the `sizeof` or `typeid` call, and when the lambda functor is called the wrapped operation is performed. For example:

```
vector<base*> a;
...
for_each(a.begin(), a.end(),
         cout << bind(&type_info::name, ll_typeid(*_1)));
```

Here `ll_typeid` creates a lambda functor for calling `typeid` for each element. The result of a `typeid` call is an instance of the `type_info` class, and the bind expression creates a lambda functor for calling the `name` member function of that class.

Nesting STL algorithm invocations

The BLL defines common STL algorithms as function object classes, instances of which can be used as target functions in bind expressions. For example, the following code iterates over the elements of a two-dimensional array, and computes their sum.

```
int a[100][200];
int sum = 0;

std::for_each(a, a + 100,
       bind(ll::for_each(), _1, _1 + 200, protect(sum += _1)));
```

The BLL versions of the STL algorithms are classes, which define the function call operator (or several overloaded ones) to call the corresponding function templates in the `std` namespace. All these structs are placed in the subnamespace `boost::lambda::ll`.

Note that there is no easy way to express an overloaded member function call in a lambda expression. This limits the usefulness of nested STL algorithms, as for instance the `begin` function has more than one overloaded definitions in container templates. In general, something analogous to the pseudo-code below cannot be written:

```
std::for_each(a.begin(), a.end(),
       bind(ll::for_each(), _1.begin(), _1.end(), protect(sum += _1)));
```

Some aid for common special cases can be provided though. The BLL defines two helper function object classes, `call_begin` and `call_end`, which wrap a call to the `begin` and, respectively, `end` functions of a container, and return the `const_iterator` type of the container. With these helper templates, the above code becomes:

```
std::for_each(a.begin(), a.end(),
       bind(ll::for_each(),
                   bind(call_begin(), _1), bind(call_end(), _1),
                        protect(sum += _1)));
```

Extending return type deduction system

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

The overloadable user defined operators are either unary or binary. For each arity, there are two traits templates that define the return types of the different operators. Hence, the return type system can be extended by providing more specializations for these templates. The templates for unary functors are `plain_return_type_1<Action, A>` and `return_type_1<Action, A>` , and `plain_return_type_2<Action, A, B>` and `return_type_2<Action, A, B>` respectively for binary functors.

The first parameter (`Action`) to all these templates is the *action* class, which specifies the operator. Operators with similar return type rules are grouped together into *action groups*, and only the action class and action group together define the operator unambiguously. As an example, the action type `arithmetic_action<plus_action>` stands for `operator+`. The complete listing of different action types is shown in Table 2, “Action types”.

The latter parameters, `A` in the unary case, or `A` and `B` in the binary case, stand for the argument types of the operator call. The two sets of templates, `plain_return_type_`*n* and `return_type_`*n* (*n* is 1 or 2) differ in the way how parameter types are presented to them. For the former templates, the parameter types are always provided as non-reference types, and do not have const or volatile qualifiers. This makes specializing easy, as commonly one specialization for each user defined operator, or operator group, is enough. On the other hand, if a particular operator is overloaded for different cv-qualifications of the same argument types, and the return types of these overloaded versions differ, a more fine-grained control is needed. Hence, for the latter templates, the parameter types preserve the cv-qualifiers, and are non-reference types as well. The downside is, that for an overloaded set of operators of the kind described above, one may end up needing up to 16 `return_type_2` specializations.

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

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

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

```
namespace boost {
namespace lambda {

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

}
}
```

Having this specialization defined, BLL is capable of correctly deducing the return type of the above two operators. Note, that the specializations must be in the same namespace, `::boost::lambda`, with the primary template. For brevity, we do not show the namespace definitions in the examples below.

It is possible to specialize on the level of an individual operator as well, in addition to providing a specialization for a group of operators. Say, we add a new arithmetic operator for argument types `X` and `Y`:

```
X operator*(const X&, const Y&);
```

Our first rule for all arithmetic operators specifies that the return type of this operator is `Z`, which obviously is not the case. Hence, we provide a new rule for the multiplication operator:

```
template<>
struct plain_return_type_2<arithmetic_action<multiply_action>, X, Y> {
  typedef X type;
};
```

The specializations can define arbitrary mappings from the argument types to the return type. Suppose we have some mathematical vector type, templated on the element type:

```
template <class T> class my_vector;
```

Suppose the addition operator is defined between any two `my_vector` instantiations, as long as the addition operator is defined between their element types. Furthermore, the element type of the resulting `my_vector` is the same as the result type of the addition between the element types. E.g., adding `my_vector<int>` and `my_vector<double>` results in `my_vector<double>`. The BLL has traits classes to perform the implicit built-in and standard type conversions between integral, floating point, and complex classes. Using BLL tools, the addition operator described above can be defined as:

```
template<class A, class B>
my_vector<typename return_type_2<arithmetic_action<plus_action>, A, B>::type>
operator+(const my_vector<A>& a, const my_vector<B>& b)
{
  typedef typename
    return_type_2<arithmetic_action<plus_action>, A, B>::type res_type;
  return my_vector<res_type>();
}
```

To allow BLL to deduce the type of `my_vector` additions correctly, we can define:

```
template<class A, class B>
class plain_return_type_2<arithmetic_action<plus_action>,
                          my_vector<A>, my_vector<B> > {
  typedef typename
    return_type_2<arithmetic_action<plus_action>, A, B>::type res_type;
public:
  typedef my_vector<res_type> type;
};
```

Note, that we are reusing the existing specializations for the BLL `return_type_2` template, which require that the argument types are references.

Table 2. Action types

+	arithmetic_action<plus_action>
-	arithmetic_action<minus_action>
*	arithmetic_action<multiply_action>
/	arithmetic_action<divide_action>
%	arithmetic_action<remainder_action>
+	unary_arithmetic_action<plus_action>
-	unary_arithmetic_action<minus_action>
&	bitwise_action<and_action>
|	bitwise_action<or_action>
~	bitwise_action<not_action>
^	bitwise_action<xor_action>
<<	bitwise_action<leftshift_action_no_stream>
>>	bitwise_action<rightshift_action_no_stream>
&&	logical_action<and_action>
||	logical_action<or_action>
!	logical_action<not_action>
<	relational_action<less_action>
>	relational_action<greater_action>
<=	relational_action<lessorequal_action>
>=	relational_action<greaterorequal_action>
==	relational_action<equal_action>
!=	relational_action<notequal_action>
+=	arithmetic_assignment_action<plus_action>
-=	arithmetic_assignment_action<minus_action>
*=	arithmetic_assignment_action<multiply_action>
/=	arithmetic_assignment_action<divide_action>
%=	arithmetic_assignment_action<remainder_action>
&=	bitwise_assignment_action<and_action>

=|	bitwise_assignment_action<or_action>
^=	bitwise_assignment_action<xor_action>
<<=	bitwise_assignment_action<leftshift_action>
>>=	bitwise_assignment_action<rightshift_action>
++	pre_increment_decrement_action<increment_action>
--	pre_increment_decrement_action<decrement_action>
++	post_increment_decrement_action<increment_action>
--	post_increment_decrement_action<decrement_action>
&	other_action<address_of_action>
*	other_action<contents_of_action>
,	other_action<comma_action>
->*	other_action<member_pointer_action>

Practical considerations

Performance

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 can. Depending on the compiler, this can also be true in practice. We ran two tests with the GCC 3.0.4 compiler on 1.5 GHz Intel Pentium 4. The optimization flag -03 was used.

In the first test we compared lambda functors against explicitly written function objects. We used both of these styles to define unary functions which multiply the argument repeatedly by itself. We started with the identity function, going up to x^5. The expressions were called inside a `std::transform` loop, reading the argument from one `std::vector<int>` and placing the result into another. The length of the vectors was 100 elements. The running times are listed in Table 3, “Test 1”. We can observe that there is no significant difference between the two approaches.

In the second test we again used `std::transform` to perform an operation to each element in a 100-element long vector. This time the element type of the vectors was `double` and we started with very simple arithmetic expressions and moved to more complex ones. The running times are listed in Table 4, “Test 2”. Here, we also included classic STL style unnamed functions into tests. We do not show these expressions, as they get rather complex. For example, the last expression in Table 4, “Test 2” written with classic STL tools contains 7 calls to `compose2`, 8 calls to `bind1st` and altogether 14 constructor invocations for creating `multiplies`, `minus` and `plus` objects. In this test the BLL expressions are a little slower (roughly 10% on average, less than 14% in all cases) than the corresponding hand-written function objects. The performance hit is a bit greater with classic STL expressions, up to 27% for the simplest expressios.

The tests suggest that the BLL does not introduce a loss of performance compared to STL function objects. With a reasonable optimizing compiler, one should expect the performance characteristics be comparable to using classic STL. Moreover, with simple expressions the performance can be expected to be close to that of explicitly written function objects. Note however, that evaluating a lambda functor consist of a sequence of calls to small functions that are declared inline. If the compiler fails to actually expand these functions inline, the performance can suffer. The running time can more than double if this happens. Although the above tests do not include such an expression, we have experienced this for some seemingly simple expressions.

Table 3. Test 1

expression	lambda expression	hand-coded function object
x	240	230
x*x	340	350
x*x*x	770	760
x*x*x*x	1180	1210
x*x*x*x*x	1950	1910

Table 4. Test 2

expression	lambda expression	classic STL expression	hand-coded function object
ax	330	370	290
-ax	350	370	310
ax-(a+x)	470	500	420
(ax-(a+x))(a+x)	620	670	600
((ax) - (a+x))(bx - (b+x))(ax - (b+x))(bx - (a+x))	1660	1660	1460

Some additional performance testing with an earlier version of the library is described [Jär00].

About compiling

The BLL uses templates rather heavily, performing numerous recursive instantiations of the same templates. This has (at least) three implications:

- While it is possible to write incredibly complex lambda expressions, it probably isn't a good idea. Compiling such expressions may end up requiring a lot of memory at compile time, and being slow to compile.
- The types of lambda functors that result from even the simplest lambda expressions are cryptic. Usually the programmer doesn't need to deal with the lambda functor types at all, but in the case of an error in a lambda expression, the compiler usually outputs the types of the lambda functors involved. This can make the error messages very long and difficult to interpret, particularly if the compiler outputs the whole chain of template instantiations.
- The C++ Standard 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 commonly require the limit explicitly increased with a command line argument.

Portability

The BLL works with the following compilers, that is, the compilers are capable of compiling the test cases that are included with the BLL:

- GCC 3.0.4
- KCC 4.0f with EDG 2.43.1
- GCC 2.96 (fails with one test case, the `exception_test.cpp` results in an internal compiler error.)

Test coverage

The following list describes the test files included and the features that each file covers:

- `bind_tests_simple.cpp` : Bind expressions of different arities and types of target functions: function pointers, function objects and member functions. Function composition with bind expressions.
- `bind_tests_simple_function_references.cpp` : Repeats all tests from `bind_tests_simple.cpp` where the target function is a function pointer, but uses function references instead.
- `bind_tests_advanced.cpp` : Contains tests for nested bind expressions, `unlambda`, `protect`, `const_parameters` and `break_const`. Tests passing lambda functors as actual arguments to other lambda functors, currying, and using the `sig` template to specify the return type of a function object.
- `operator_tests_simple.cpp` : Tests using all operators that are overloaded for lambda expressions, that is, unary and binary arithmetic, bitwise, comparison, logical, increment and decrement, compound, assignment, subscrict, address of, dereference, and comma operators. The streaming nature of shift operators is tested, as well as pointer arithmetic with plus and minus operators.
- `member_pointer_test.cpp` : The pointer to member operator is complex enough to warrant a separate test file.
- `control_structures.cpp` : Tests for the looping and if constructs.
- `switch_construct.cpp` : Includes tests for all supported arities of the switch statement, both with and without the default case.
- `exception_test.cpp` : Includes tests for throwing exceptions and for try/catch constructs with varying number of catch blocks.
- `constructor_tests.cpp` : Contains tests for `constructor`, `destructor`, `new_ptr`, `delete_ptr`, `new_array` and `delete_array`.
- `cast_test.cpp` : Tests for the four cast expressions, as well as `typeid` and `sizeof`.
- `extending_return_type_traits.cpp` : Tests extending the return type deduction system for user defined types. Contains several user defined operators and the corresponding specializations for the return type deduction templates.
- `is_instance_of_test.cpp` : Includes tests for an internally used traits template, which can detect whether a given type is an instance of a certain template or not.
- `bll_and_function.cpp` : Contains tests for using `boost::function` together with lambda functors.

Relation to other Boost libraries

Boost Function

Sometimes it is convenient to store lambda functors in variables. However, the types of even the simplest lambda functors are long and unwieldy, and it is in general unfeasible to declare variables with lambda functor types. *The Boost Function library* [function] defines wrappers for arbitrary function objects, for example lambda functors; and these wrappers have types that are easy to type out. For example:

```
boost::function<int(int, int)> f = _1 + _2;
boost::function<int&(int&)> g = (_1 += 10);
int i = 1, j = 2;
f(i, j); // returns 3
g(i);    // sets i to = 11;
```

The return and parameter types of the wrapped function object must be written explicilty as the template argument to the wrapper template `boost::function`; even when lambda functors, which otherwise have generic parameters, are wrapped. Wrapping a function object with `boost::function` introduces a performance cost comparable to virtual function dispatch, though virtual

functions are not actually used. Note that storing lambda functors inside `boost::function` introduces a danger. Certain types of lambda functors may store references to the bound arguments, instead as taking copies of the arguments of the lambda expression. When temporary lambda functor objects are used in STL algorithm invocations this is always safe, as the lambda functor gets destructed immediately after the STL algortihm invocation is completed. However, a lambda functor wrapped inside `boost::function` may continue to exist longer, creating the possibility of dangling references. For example:

```
int* sum = new int();
*sum = 0;
boost::function<int&(int)> counter = *sum += _1;
counter(5); // ok, *sum = 5;
delete sum;
counter(3); // error, *sum does not exist anymore
```

Boost Bind

The Boost Bind [bind] library has partially overlapping functionality with the BLL. Basically, the Boost Bind library (BB in the sequel) implements the bind expression part of BLL. There are, however, some semantical differerences.

The BLL and BB evolved separately, and have different implementations. This means that the bind expressions from the BB cannot be used within bind expressions, or within other type of lambda expressions, of the BLL. The same holds for using BLL bind expressions in the BB. The libraries can coexist, however, as the names of the BB library are in `boost` namespace, whereas the BLL names are in `boost::lambda` namespace.

The BLL requires a compiler that is reasonably conformant to the C++ standard, whereas the BB library is more portable, and works with a larger set of compilers.

The following two sections describe what are the semantic differences between the bind expressions in BB and BLL.

First argument of bind expression

In BB the first argument of the bind expression, the target function, is treated differently from the other arguments, as no argument substitution takes place within that argument. In BLL the first argument is not a special case in this respect. For example:

```
template<class F>
int foo(const F& f) {
  int x;
  ..
  bind(f, _1)(x);
  ...
}
```

```
int bar(int, int);
nested(bind(bar, 1, _1));
```

The bind expression inside `foo` becomes:

```
bind(bind(bar, 1, _1), _1)(x)
```

The BLL interpretes this as:

```
bar(1, x)(x)
```

whereas the BB library as

```
bar(1, x)
```

To get this functionality in BLL, the bind expression inside the `foo` function can be written as:

```
bind(unlambda(f), _1)(x);
```

as explained in the section called “Unlambda”.

The BB library supports up to nine placeholders, while the BLL defines only three placeholders. The rationale for not providing more, is that the highest arity of the function objects accepted by any STL algorithm is two. The placeholder count is easy to increase in the BB library. In BLL it is possible, but more laborous. The BLL currently passes the actual arguments to the lambda functors internally just as they are and does not wrap them inside a tuple object. The reason for this is that some widely used compilers are not capable of optimizing the intermediate tuple objects away. The creation of the intermediate tuples would cause a significant performance hit, particularly for the simplest (and thus the most common) lambda functors. We are working on a hybrid approach, which will allow more placeholders but not compromise the performance of simple lambda functors.

Contributors

The main body of the library was written by Jaakko Järvi and Gary Powell. We've got outside help, suggestions and ideas from Jeremy Siek, Peter Higley, Peter Dimov, Valentin Bonnard, William Kempf. We would particularly like to mention Joel de Guzmann and his work with Phoenix which has influenced BLL significantly, making it considerably simpler to extend the library with new features.

Rationale for some of the design decisions

Lambda functor arity

The highest placeholder index in a lambda expression determines the arity of the resulting function object. However, this is just the minimal arity, as the function object can take arbitrarily many arguments; those not needed are discarded. Consider the two bind expressions and their invocations below:

```
bind(g, _3, _3, _3)(x, y, z);
bind(g, _1, _1, _1)(x, y, z);
```

This first line discards arguments `x` and `y`, and makes the call:

```
g(z, z, z)
```

whereas the second line discards arguments `y` and `z`, and calls:

```
g(x, x, x)
```

In earlier versions of the library, the latter line resulted in a compile time error. This is basically a tradeoff between safety and flexibility, and the issue was extensively discussed during the Boost review period of the library. The main points for the *strict arity*

checking was that it might catch a programming error at an earlier time and that a lambda expression that explicitly discards its arguments is easy to write:

```
(_3, bind(g, _1, _1, _1))(x, y, z);
```

This lambda expression takes three arguments. The left-hand argument of the comma operator does nothing, and as comma returns the result of evaluating the right-hand argument we end up with the call `g(x, x, x)` even with the strict arity.

The main points against the strict arity checking were that the need to discard arguments is commonplace, and should therefore be straightforward, and that strict arity checking does not really buy that much more safety, particularly as it is not symmetric. For example, if the programmer wanted to write the expression `_1 + _2` but mistakenly wrote `_1 + 2`, with strict arity checking, the complier would spot the error. However, if the erroneous expression was `1 + _2` instead, the error would go unnoticed. Furthermore, weak arity checking simplifies the implementation a bit. Following the recommendation of the Boost review, strict arity checking was dropped.

Bibliography

[STL94] A. A. Stepanov and M. Lee. *The Standard Template Library*. Hewlett-Packard Laboratories. 1994. www.hpl.hp.com/techreports .

[SGI02] *The SGI Standard Template Library*. 2002. www.sgi.com/tech/stl/.

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

[Järr99] Jaakko Järvi. *C++ Function Object Binders Made Easy*. . *Lecture Notes in Computer Science*. 1977. Springer. 2000.

[Järr00] Jaakko Järvi. Gary Powell. *The Lambda Library : Lambda Abstraction in C++*. Turku Centre for Computer Science. Technical Report . 378. 2000. www.tucs.fi/publications.

[Järr01] Jaakko Järvi. Gary Powell. *The Lambda Library : Lambda Abstraction in C++*. Second Workshop on C++ Template Programming. Tampa Bay, OOPSLA'01. . 2001. www.oonumerics.org/tmpw01/.

[Järr03] Jaakko Järvi. Gary Powell. Andrew Lumsdaine. *The Lambda Library : unnamed functions in C++*. . *Software - Practice and Expreience*. 33:259-291. 2003.

[tuple] *The Boost Tuple Library*. www.boost.org/libs/tuple/doc/tuple_users_guide.html . 2002.

[type_traits] *The Boost type_traits*. www.boost.org/libs/type_traits/ . 2002.

[ref] *Boost ref*. www.boost.org/libs/bind/ref.html . 2002.

[bind] *Boost Bind Library*. www.boost.org/libs/bind/bind.html . 2002.

[function] *Boost Function Library*. www.boost.org/libs/function/ . 2002.

[fc++] *The FC++ library: Functional Programming in C++*. Yannis Smaragdakis. Brian McNamara. www.cc.gatech.edu/~yannis/fc++/ . 2002.