# Using TR1's bind with Containers and Algorithms

**David Kieras, EECS Department, University of Michigan**

**February, 2007**

Since the first C++ Standard was approved in 1998, a group of C++ wizards have been working on the open-source Boost library (www.boost.org), with the goal of developing new library facilities good enough to be considered for addition to the C++ Standard Library. The Standard Library Technical Report No. 1 (TR1) specifies several Boost libraries that are scheduled to become part of the official C++ Standard Library in the next version of the Standard (expected to be completed in 2009). This note deals with a single facility, `bind`, that in one function template replaces several more limited and clumsy adapters and binders in the Standard Library, namely `ptr_fun, mem_fun, mem_fun_ref, bind1st,` and `bind2nd`. The bind facility can be used in combination with algorithms such as `for_each` to apply a function to each item in a container. This document is a tutorial on the bind facility and how to use it with the Standard Library algorithms. See the posted code examples for more complete examples than presented here.

## Basics of bind

### How to access tr1::bind

gcc 4.x comes with Boost's version of the TR1 library. The following #includes and using directives make the facilities described in this handout available in gcc 4.x. Other compiler/library environments will differ.

```
#include <tr1/functional>

using namespace std::tr1;
using namespace std::tr1::placeholders;
```

### Creating a function object with bound arguments

The TR1 `bind` template is an elaborate function template that creates and returns a function object that "wraps" a supplied function in the form of a function pointer. Its basic form is:

```
bind(function pointer, bound arguments)
```

For example, let sum3 be a function that takes 3 integer arguments and returns an int that is the sum of its arguments:

```
int sum3(int x, int y, int z)
{
   int sum = x+y+z;
   cout << x << '+' << y << '+' << z << '=' << sum << endl;
   return sum;
}
```

Let int1, int2, and int3 be three int variables. Then

```
bind(&sum3, int1, int2, int3)
```

creates and returns a function object whose function call operator takes no arguments and calls sum3 with the values in the three integer variables, and returns an int. We can invoke the function call operator with a call with no additional arguments, as in:

```
int result = bind(&sum3, int1, int2, int3)();
```

The final effect is as if sum3 was called as:

```
int result = sum3(int1, int2, int3);
```

However, note that the values in the bound arguments are copied into the function object and stored as member variable values at the time the function object is created. Thus bind acts to "bind" a function to a set of argument values, and produce a function object that packages the stored values together with a pointer to the function, and enables the function to be called with fewer arguments - in this case none because all of the function arguments are bound to the supplied values.

To try to avoid confusion, the actual arguments required by the wrapped function (sum3 in this case) will be called the *function arguments* to distinguish them from the *bound arguments* supplied as additional arguments to the bind function template. A basic rule is that the number of bound arguments has to equal the number of function arguments, and the order of them corresponds - the first bound argument corresponds to the first function argument, the second to the second, and so forth.

The bound arguments can also have literal values, as in:

```
bind(&sum3, 42, int2, int3)();
```

The bound values are copied into the function object, so if the function takes a call-by-reference argument and modifies it, the modification will be to internal copy in the function object, and not the original variable. However, TR1 includes a *reference wrapper*, invoked with the `ref` function template, which produces the effect of a copyable reference. If you wrap one of the bound arguments in a reference wrapper, and the function modifies it, the actual bound variable will get modified.

For example, suppose function mod23 takes its second and third int argument by reference and modifies them:

```
void mod23(int x, int & y, int & z)
{
   y = y + x;
   z = z + y;
}
```

Then

```
cout << int1 << ' ' << int2 << ' ' << int3 << endl;
bind(& mod23, int1, int2, ref(int3))();
cout << int1 << ' ' << int2 << ' ' << int3 << endl;
```

would result in the same value for int1 and int2 in the two output statements, but a different value for int3.

As a further example, suppose we have a function that takes a stream reference argument:

```
void write_int(ostream& os, int x)
{
   os << x << endl;
}
```

We can use a reference wrapper to hand the stream in by reference in a bind call:

```
bind(&write_int, ref(cout), _1)(int1);
```

The current Standard binders and adapters won't allow a non-const reference argument, so this is a significant improvement!

### Mixing bound and call arguments with placeholders

Now we come to the most interesting part. At the time we use the function object in a call, the final function arguments can taken from a mixture of bound arguments and the arguments supplied in the call, which will be termed the c*all arguments*. This is done with special *placeholders* in the list of bound arguments.

If one of the bound arguments is a placeholder, the corresponding function argument is taken from the call arguments. For example, the following would result in a call to sum3 with the same input as the above examples.

```
bind(&sum3, _2, int2, _1)(int3, int1);
```

The placeholders in the bound argument list are the special symbols `_2` and `_1`. The call arguments are listed in the second set of parentheses, which is simply the normal syntax for an argument list in a function call,

The placeholder notation requires some care to understand. The placeholder indicates which function argument should be filled with which value from the call argument list. The *position* of the placeholder in the bound argument list corresponds to the same position in the function argument list, and the *number* of the placeholder corresponds to a value in the call argument list.

Thus, in the above example, the placeholder _2 appears first in the bound argument list, which means it specifies the first argument to be given to sum3. Its number, 2, specifies the second argument in the call argument list. Thus the second call argument item will be the first function argument. Likewise, _1 in the third position means that this argument to sum3 should be taken from the first of the supplied values in the call arguments. In this case, bind creates a function object that can be called with two integer values, the first of which is given to sum3 as its third argument, the second of which is given to sum3 as its first argument, and the second argument given to sum3 is the bound value of int2, which is copied and stored when the function object is created.

Any mixture of placeholders and ordinary arguments can appear, as long as the number of items in the bound argument list equals the number of function arguments. In contrast, the call argument list can include extra values - they don't all have to be used, as long as the wrapped function gets all the arguments it needs. Likewise, it can include fewer values if some of them get used more than once. To illustrate the extremes, we could write both of the following:

```
bind(&sum3, _2, int2, _5)(int3, int1, int4, int5, int6);
bind(&sum3, _1, _1, _1)(int1);
```

The first call uses the second and fifth call arguments and ignores the others. The second call uses the same call argument for all three function arguments.

The call arguments are passed into the function call by non-const reference, meaning that the function can modify the original values if it has a call-by-reference parameter. Since mod23 modifies its second and third arguments, if we write:

```
cout << int1 << ' ' << int2 << ' ' << int3 << endl;
bind(&mod23, int1, _1, _2)(int2, int3);
cout << int1 << ' ' << int2 << ' ' << int3 << endl;
```

Both int2 and int3 will have different values in the two output statements.

A consequence of handling the call arguments as non-const reference is that unlike the bound arguments, the call arguments cannot be literal values - they have to be variables. This constraint is due to the "forwarding problem" in the current Standard C++. This might change if some language fundamentals are changed in the future.

### Using bind in algorithms

All the background has now been presented for how to use bind in the context of an algorithm like for_each running over a container of objects. When the algorithm is executed, the dereferenced iterator has a single value, and this single value will constitute the single call argument to the bind function object.

We can use bind to bind a function that takes many arguments to values for all but one of the arguments, and have the value for this one argument be supplied by the dereferenced iterator. This usually means that only the placeholder _1 will appear in the bound arguments, because there is only one value in the call argument list. The position of _1 in the bound argument list corresponds to which parameter of the wrapped function we want to come from the iterator.

Suppose int_list is a std::list of integers, then:

```
for_each(int_list.begin(), int_list.end(), bind(&sum3, _1, 5, 9) );
```

will apply the sum3 function to each integer in the list, with the first argument being the dereferenced iterator value, as shown by the placeholder, and the constants 5 and 9 being the second and third.

The following will apply the mod23 function, with the list item being the third parameter, and will result in the modified values being copied both into the bind function object for the second parameter, and back into the list for the third parameter.

```
for_each(int_list.begin(), int_list.end(), bind(&mod23, 3, 5, _1) );
```

By using the reference wrapper, we can call a function that takes a stream object by reference as one argument:

```
for_each(int_list.begin(), int_list.end(), bind(&write_int, ref(cout), _1) );
```

## Using bind with Class Objects

### A simple example class and functions

Here is presented an simple class, called Thing, that will be used in the examples in the remainder of this handout. Thing contains an integer specified in its constructor, and has simple const member functions for printing and non-const member functions for modifying the internal value. These functions take either 0, 1, or 2 arguments, but remember that member functions have a hidden parameter for the "this" pointer. Thus, for purposes of bind, these member functions have 1, 2, or 3 parameters.

```
class Thing {
public:
    Thing(int in_i = 0) : i(in_i) {}
    void print() const  // a const member function
        {cout << "Thing " << i << endl;}
    void write(ostream& os) const  // write to a supplied ostream
        {os << "Thing" << i << " written to stream" << endl;}
    void print1arg(int j) const // a const member function with 1 argument
        {cout << "Thing " << i << " with arg " << j << endl;}
    void print2arg(int j, int k) const   // with 2 arguments
        {cout << "Thing " << i << " with args " << j << ' ' << k << endl;}
    void update() // a modifying function with no arguments
        {i++; cout << "Thing updated to " << i << endl;}
    void set_value(int in_i)   // a modifying function with one argument
        {i = in_i; cout << "Thing value set to " << i << endl;}
private:
    int i;
};

ostream& operator<< (ostream& os, const Thing& t)
{
    os << "Thing: " << t.get_value();
    return os;
}
```

We also have some non-member functions to play with:

```
void print_Thing(Thing t)
    {t.print();}

void print_Thing_const_ref(const Thing& t)
    {t.print();}

void print_int_Thing(int i, Thing t)
    {cout << "print_int_Thing " << i << ' ' << t << endl;}
```

```
void print_Thing_int(Thing t, int i)
   {cout << "print_Thing_int " << t << ' ' << i << endl;}

void print_Thing_int_int(Thing t, int i, int j)
   {cout << "print_Thing_int_int " << t << ' ' << i << ' ' << j << endl;}
```

### Class object arguments and member functions

Non-member function calls involving a Thing object are just like the above examples for functions that take an integer, where we can specify the object or other arguments as either bound arguments or call arguments:

```
int int1 = 42;
int int2 = 76;
int int3 = 88;
Thing t1(1);

bind(&print_Thing, t1)();
bind(&print_Thing_const_ref, t1)();
bind(&print_int_Thing, int1, t1)();

bind(&print_Thing, _1)(t1);
bind(&print_Thing_const_ref, _1)(t1);
bind(&print_int_Thing, _1, _2)(int1, t1);
```

We can call functions that modify the supplied object, but we need to consider whether the modified object is one copied and stored in the function object, a reference-wrapped object, or an object in the call arguments:

```
bind(&update_Thing, t1)();           // modify a copy of t1
bind(&update_Thing, ref(t1))();      // modify original t1
bind(&update_Thing, _1)(t1);         // modify original t1

bind(&set_Thing, t1, int1)();        // modify a copy of t1
bind(&set_Thing, ref(t1), int2)();   // modify original t1
bind(&set_Thing, _1, _2)(t1, int3);  // modify original t1
```

Calls to member functions are just as simple, because the bind template is "smart" enough to automatically figure out that a pointer-to-member-function is involved and how to set up the call to it. We just need to make sure that a Thing object is supplied as the actual first function argument to be "this" object. It works for both const and non-const member functions:

```
bind(&Thing::print, _1)(t1);
bind(&Thing::write, _1, ref(cout))(t1);
bind(&Thing::print1arg, _1, _2)(t1, int2);
bind(&Thing::print2arg, _2, _1, _3)(int3, t1, int2);

bind(&Thing::update, _1)(t1);           // modify original t1
bind(&Thing::set_value, _1, int1)(t1);  // modify original t1
bind(&Thing::update, t1)();             // modify a copy of t1
bind(&Thing::update, ref(t1))();        // modify original t1
```

## Using bind with Algorithms on Containers of Class Objects

### Sequence containers of objects

We can use for_each to apply various non-member and member functions to each Thing in the container, exactly along the lines already described for a container of ints. Let's start with a list of Things and non-member functions that don't modify the object:

```
int int1 = 42;
int int2 = 76;
Thing t1(1), t2(2), t3(3);
typedef list<Thing> olist_t;
olist_t obj_list;
obj_list.push_back(t1);
obj_list.push_back(t2);
obj_list.push_back(t3);

for_each(obj_list.begin(), obj_list.end(), bind(&print_Thing, _1) );
for_each(obj_list.begin(), obj_list.end(),
   bind(&print_Thing_int_int, _1, int1, int2) );
```

We can call non-member functions that modify the Thing in the list:

```
for_each(obj_list.begin(), obj_list.end(), bind(&update_Thing, _1) );
for_each(obj_list.begin(), obj_list.end(), bind(&set_Thing, _1, int1) );
```

As before, calling member functions only requires that we use a member function pointer and ensure that the first function argument is "this" object from the dereferenced iterator:

```
for_each(obj_list.begin(), obj_list.end(), bind(&Thing::print, _1) );
for_each(obj_list.begin(), obj_list.end(), bind(&Thing::write, _1, ref(cout)) );
for_each(obj_list.begin(), obj_list.end(),
   bind(&Thing::print2arg, _1, int1, int2) );

for_each(obj_list.begin(), obj_list.end(), bind(&Thing::update, _1) );
for_each(obj_list.begin(), obj_list.end(), bind(&Thing::set_value, _1, int1) );
```

Because bind is smart enough to automatically take into account the type of the dereferenced iterator, using a list of pointers to Thing looks exactly the same:

```
typedef list<Thing *> plist_t;
plist_t ptr_list;
ptr_list.push_back(&t1);
ptr_list.push_back(&t2);
ptr_list.push_back(&t3);

for_each(ptr_list.begin(), ptr_list.end(), bind(&Thing::print, _1) );
for_each(ptr_list.begin(), ptr_list.end(), bind(&Thing::write, _1, ref(cout)) );
for_each(ptr_list.begin(), ptr_list.end(),
   bind(&Thing::print2arg, _1, int1, int2) );

for_each(ptr_list.begin(), ptr_list.end(), bind(&Thing::update, _1) );
for_each(ptr_list.begin(), ptr_list.end(), bind(&Thing::set_value, _1, int1) );
```

The same basic approach can be used for any sequence container and any of the algorithms. The bind template is really pretty handy - it replaces the Standard ptr_fun, mem_fun, mem_fun_ref, bind1st and bind2d adapters and binders with a single template function that does it all!

### Using bind with map containers

The map container is amazingly useful but is exasperatingly clumsy with the algorithms. The Standard Library has no adapters and binders that allow you to pick out the first or second of the pair supplied by the dereferenced iterator. Consequently, if you want to use the algorithms, you often have to write a custom function or function object to pick out the pair member that you want and operate on it; if you don't want to write the custom function object, then you have to write explicit loops instead of using the algorithms.

The situation is much better with bind; in many cases it will work perfectly in allowing you to use an algorithm to iterate over a map container and apply a function to only one member of the pair. Unfortunately, the syntax involved is ugly; but it is also instructive because it illustrates how you can *compose* functions: bind can take the function object returned by a nested bind as one of its arguments.

That is, since bind creates a function object whose function call operator returns a value, one bind can serve as a value for another bind. For example,

```
bind(&sum3, int1, bind(&sum3, _1, int2, int3), int3)(x);
```

creates and calls a function object whose operator() takes one int call argument, the int variable x, and first calls sum3 with the other two arguments being the bound arguments int2 and int3. The resulting value is used as the second argument in another call to sum3, with the first being int1 and the third being int3. The resulting effect is that of a *composed* function call:

```
sum3(int1, sum3(x, int2, int3), int3);
```

Of course, different functions with different number of arguments can be called. The rule is that all the inner binds are evaluated before the outer bind. The placeholders can appear in either the inner or outer binds.

To use bind with a map container involves first using a nested bind to pick out the second of the pair from the dereferenced iterator, and then another bind to call the function with the value picked out from the pair. How is this possible?

Well, bind is extremely smart about making use of the function pointer, and can understand a pointer-to-member-function. In fact, it can make sense of something that isn't a function pointer in the usual sense of the word, but is the rarely-used *pointer-to-member-variable*. If you supply a pointer-to-member-variable, bind will construct a function object that simply returns the value of that member variable for a supplied object. The catch is that for deep technical reasons, the bind template can't deduce the type of the member variable on its own, adding a further complication. Sounds messy? The easiest way to explain is with an example using a map from int to Thing objects:

```
Thing t1(1), t2(2), t3(3);
typedef map<int, Thing> omap_t;  //typedef for clarity
omap_t obj_map;
obj_map[1] = t1; // a COPY of t1 is in the container!
obj_map[2] = t2;
obj_map[3] = t3;
```

Suppose we start to write a for_each loop that we want to apply the print member function for each Thing in the container:

```
for_each(obj_map.begin(), obj_map.end(),
   bind(&Thing::print, what goes in here?) );
```

The dereferenced iterator from the map<int, Thing> container will have a value of pair<const int, Thing>. We'll use the wacky ability of bind to construct a function object that will return the value of the second of the iterator pair, namely a copy of the Thing stored at that point in the map. This bind will look like:

```
bind<Thing>(&map<int, Thing>::value_type::second, _1)
```

Let's take this one bit at a time. The template argument `<Thing>` appearing after `bind` specifies the return type for the function object; for deep technical reasons, the bind template can't deduce this type on its own for a pointer-

to-member-variable. Next is the oddity in bind's “function pointer” slot. Recall that `value_type` is a Standard typedef for containers giving the type of object stored in the container, in this case, a pair<const int, Thing>. Also recall that, `second` is the name for the second member variable in the pair. So the “function pointer” supplied to bind is a pointer to the `second` member variable of the pair. Finally, the _1 designates the first (and only) call argument, which will be a pair from the dereferenced iterator. When the function object is called, the “function” is applied to the supplied pair, and the result is the value of the second of the pair, in this case a Thing. Ouch! Does your head hurt? Mine does!

This function object can be used as a bound argument for an outer bind that wraps the member function of Thing:

```
bind(&Thing::print, bind<Thing>(&map<int, Thing>::value_type::second, _1))
```

This calls the print member function on the Thing object copied from the second of the supplied pair. This whole mess can be used in a for_each algorithm to call the print member function for the second of each pair in a map container. This is shown below with some indentation and using our map typedef name to assist reading:

```
for_each(obj_map.begin(), obj_map.end(),
   bind(&Thing::print,
      bind<Thing>(&omap_t::value_type::second, _1)) );
```

While the inner bind is horrible, it is the only horrible thing! To get the first of the pair, just substitute the appropriate type in the bind template parameter and `first` instead of `second` in the "function pointer" slot.

We can throw in extra arguments to the member function along the previous lines, as long as we keep straight that the position in the bound argument list corresponds to the function parameters, and the first parameter is the “this” argument supplied as the second of the pair. The following calls print2arg for each Thing in the map:

```
for_each(obj_map.begin(), obj_map.end(),
   bind(&Thing::print2arg,
      bind<Thing>(&omap_t::value_type::second, _1), int1, int2) );
```

We can also hand in a stream parameter by reference:

```
for_each(obj_map.begin(), obj_map.end(),
   bind(&Thing::write,
      bind<Thing>(&omap_t::value_type::second, _1), ref(cout)) );
```

### There’s gotta be a catch ...

So far, everything looks pretty easy, once we get used to that crazy bind of the “function” second. However, there is a problem waiting for us. The inner bind creates and returns a function object that can be immediately called and returns a value. Its returned value is a temporary object, and is copied and stored as a bound argument, and is then supplied as an input function argument for the wrapped function. Here's the gotcha: You can't modify a temporary object with a non-member function that takes a non-const reference parameter, or a non-const member function that takes the temporary object as "this" object.

In other words, if you have a map container of objects, the inner bind that picks out the second of the pair gives you something you can’t modify. So you can't use this mechanism to invoke a function that will modify the objects in the container! To make this clear with our example:

```
// compile error: initializing non-const ref to a temporary
for_each(obj_map.begin(), obj_map.end(),
   bind(update_Thing,
      bind<Thing>(&omap_t::value_type::second, _1)) );

// compile error: no match for call of non-const member function
for_each(obj_map.begin(), obj_map.end(),
   bind(&Thing::update,
      bind<Thing>(&omap_t::value_type::second, _1)) );
```

### But bind with maps of pointers to objects is well-behaved!

Before you conclude that reading this handout has been a complete waste of time, first, remember that the problem does not appear with sequence containers of objects or pointers to objects, because the embedded bind to pick out the second of the pair is not needed.

Second, for map containers, the problem does not appear if the container holds *pointers* to objects, because it is the pointer that gets passed as the temporary bound argument, not the object itself. Since you never modify the "this" pointer anyway, there is no problem, and you can modify the pointed-to objects freely. The syntax is just as simple (or no worse)!

Thus, if you have a map of pointers, you can use bind to apply modifying functions to the pointed-to objects:

```
typedef map<int, Thing *> pmap_t; //typedef for clarity
pmap_t ptr_map;
ptr_map[1] = &t1;
ptr_map[2] = &t2;
ptr_map[3] = &t3;

for_each(ptr_map.begin(), ptr_map.end(),
   bind(&Thing::update,
      bind<Thing *>(&pmap_t::value_type::second, _1)) );
for_each(ptr_map.begin(), ptr_map.end(),
   bind(&Thing::set_value,
      bind<Thing *>(&pmap_t::value_type::second, _1), int1) );
// show the result
for_each(ptr_map.begin(), ptr_map.end(),
   bind(&Thing::print,
      bind<Thing *>(&pmap_t::value_type::second, _1)) );
```

Note that we can use an additional Standard map<> typedef for the second of the pair to automate writing this code a tad more:

```
for_each(ptr_map.begin(), ptr_map.end(),
   bind(&Thing::print,
      bind<pmap_t::mapped_type>(&pmap_t::value_type::second, _1)) );
```

This way the only thing you need to change from one specific map to another is the map typedef name and the map container name - you could memorize the rest of the statement pretty quickly.

### A handy short-cut if no binding is needed

TR1 also includes a function template called `mem_fn` (note the spelling) which replaces the the `mem_fun` and `mem_fun_ref` adapters in the Standard Library. It uses the same sophisticated template programming as bind, and so is “smarter” than the previous Standard Library facilities. This one adapter works for both containers of objects and containers of pointers. Like bind, mem_fun creates and returns a function object that can be used with function call syntax to call the the wrapped function, both for a supplied object and a pointer to an object.

```
mem_fn(&Thing::print) (t1);
mem_fn(&Thing::print) (t1_ptr);
```

The first line calls Thing::print for the supplied object; the second for the object being pointed to by the supplied pointer. Notice how mem_fun is able to figure out how to do the call from the type of the supplied argument, so the syntax is identical in both cases. When used in an algorithm like for_each, the dereferenced iterator value will be the argument suppled to the function object to play the role of “this” object:

```
for_each(obj_list.begin(), obj_list.end(), mem_fn(&Thing::print));
```

If you don’t need to bind any arguments, mem_fn will do the job even more easily than bind.

## Conclusion

If you use sequence containers, read-only map containers of objects, or map containers of pointers, then bind and mem_fn will make using algorithms easy and fun!

## Where to Read More

To read more about TR1 and the Boost libraries, try the documentation on the boost web site; it ranges from quite cryptic to very clear and useful. The following books were available at the time of this writing:

Becker, P. *The C++ Standard Library Extensions: A Tutorial and Reference*. Addison-Wesley, 2007. Becker is a widely-respected C++ expert, specializing in library implementations. This book gets into how TR1 is implemented more than just how to use it, and so is pretty difficult.

Karlsson, Bjorn. *Beyond the C++ Standard Library; An Introduction to Boost*. This book surveys a large part of the Boost library from the perspective of why and how you should use it to improve your code. TR1 is well covered. Fairly clear and practical, but beware: not all of the examples will compile in gcc's version of TR1, so be sure to try out his exact examples and get them working before going on to use the facilities.