Using C++11’s `bind` with Containers and Algorithms

David Kieras, EECS Department, University of Michigan

February, 2013

This note deals with C++11’s `bind`, that in one function template replaces several more limited and clumsy adapters and binders in the C++98 Standard Library, namely `ptr_fun`, `mem_fun`, `mem_fun_ref`, `bind1st`, and `bind2nd`. The new `bind` is so much better that using these old facilities should be phased out. Accordingly, they are deprecated in the C++11 Standard, which means that while they are currently part of Standard C++, they may be removed in a future revision of the Standard. This is a strong hint to quit using them in new code!

In a nutshell, `bind` creates a function object that contains a pointer to a function and member variables that store some values for function parameters, and provides a function call operator that takes any remaining parameters and calls the pointed-to function with them along with the stored values. A common use of `bind` is to create function objects that can be used in algorithms such as `for_each` to apply a function to each item in a container. This document is a tutorial on `bind` and how to use it with the Standard Library algorithms and containers. See the posted code examples for more complete examples than presented here.

Basics of `bind`

How to access `bind`

Assuming you have a C++11 compiler and Standard Library, the following #includes and using directives make the facilities described in this handout available.

```cpp
#include <functional>
// the following are for convenience in this handout code
using namespace std;
using namespace std::placeholders;
```

Creating a function object with bound arguments

The `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:

```cpp
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 in the function call argument list, 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);

When the function object is created, a pointer to the function is stored in a member variable, and the values in the bound arguments are copied into the function object and stored as member variable values. When the function call operator is executed, the saved pointer is used to call the function, and the saved values are supplied as arguments to that function. Thus bind acts to “bind” a function to a set of argument values, and produces 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 example, none because all of the function arguments are bound to the supplied values.

To try to avoid confusion in what follows, 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 the 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 be literal values, as in:

`bind(sum3, 10, int2, int3)();`
`bind(sum3, 10, 20, 30)();`

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 stored in the function object, and not the original variable. However, C++11 includes a nifty reference wrapper class, created 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 argument by reference and modifies them:

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

Then this call

`bind(mod23, int1, int2, ref(int3))();`

results in unchanged values for int1 and int2, but a different value for int3.

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

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

We can use a reference wrapper to hand the stream in by reference to the wrapped function:

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

**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 call 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`. To make it possible to avoid name collisions, these are in the special namespace `std::placeholders`. The `using` directive above allows us to refer to them directly, instead of say `std::placeholders::_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 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:

```cpp
bind(sum3, _2, int2, _5)(int3, 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.

Suppose the function takes a call-by-reference parameter. If the corresponding argument is a modifiable location (an `lvalue`, like a non-const variable), the function can modify it. Since `mod23` modifies its second and third arguments, if we write:

```cpp
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.

We can also use constants as the call arguments, as long as the function doesn't try to modify them:

```cpp
result = bind(sum3, _1, _2, _3)(100, 200, 300);
bind(mod23, int1, _1, _2)(100, 200); // compile error!
```

We can also use `rvalues` such as expressions or function calls as the call arguments, as long as the function doesn't try to modify them:

```cpp
result = bind(sum3, _1, _2, _3)(foo(), int1+3, 300);
bind(mod23, _1, int2, _2)(foo(), int3);
bind(mod23, int1, _1, _2)(foo(), int1+3); // compile error!
```

**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<int>; then:

```cpp
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.

```cpp
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:

```cpp
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**

First we need a 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.

```cpp
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;
};
```

```cpp
ostream& operator<< (ostream& os, const Thing& t)
{
    os << "Thing: " << t.get_value();
    return os;
}
```
We also have some non-member functions to play with:

```cpp
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; }
```

### Binding 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:

```cpp
int int1 = 100;
int int2 = 200;
int int3 = 300;
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:

```cpp
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. Unlike with the ordinary function pointer, we have to use the & to specify the pointer-to-member function. We also need to make sure that a Thing object is supplied as the actual first function argument to be "this" object, so that it occupies the spot for the normally hidden this parameter of a member function. It works for both const and non-const member functions:

```cpp
bind(&Thing::print, t1)();
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:

```cpp
int int1 = 42;
int int2 = 76;
Thing t1(1), t2(2), t3(3);
typedef list<Thing> Olist_t;
Olist_t obj_list = {t1, t2, t3}; // C++11 container initialization
```

```cpp
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:

```cpp
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:

```cpp
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) );
```

```cpp
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:

```cpp
typedef list<Thing *> Plist_t;
Plist_t ptr_list = {&t1, &t2, &t3};
```

```cpp
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) );
```

```cpp
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, the set container, the unordered_set container, and any of the algorithms.
**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 lambda, a custom function, or custom function object to pick out the pair member that you want and operate on it; otherwise you have to write explicit loops instead of using the algorithms.

The situation is much better with bind; 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 function objects: 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. An example from above:

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

This creates and calls a function object whose operator() takes one call argument, the 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:

```cpp
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, and apply to the call arguments, just like in a single-level bind.

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? The easiest way to explain is with an example using a map from int to Thing objects:

```cpp
Thing t1(1), t2(2), t3(3);
ttypedef 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:

```cpp
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>. Here's the trick: bind is extremely smart about making use of a 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. We'll use this wacky ability of bind to construct a function object that will extract the second of the iterator pair, namely the Thing stored at that point in the map. This bind will look like:

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

Let's take this one bit at a time. First 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.

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

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

This calls the print member function on the Thing object extracted from the second of the supplied pair. Ouch! Does your head hurt? Mine does! 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(&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 first instead of second in the "function pointer" slot.

We can call a modifying member function on each object; the bind template deduces that it needs to provide a reference to the second of the pair to allow the object to be modified:

```
for_each(obj_map.begin(), obj_map.end(),
    bind(&Thing::update,
        bind(&Omap_t::value_type::second, _1)));
```

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(&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(&Omap_t::value_type::second, _1), ref(cout)));
```

**Using map containers of pointers to class objects**

If you have a map of pointers, you can use bind to apply functions to the pointed-to objects in the same ways as for maps of objects; you can write basically the same code, and the C++11 version of the bind template will figure things out:

```
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;

// increment the internal value
for_each(ptr_map.begin(), ptr_map.end(),
    bind(&Thing::update,
        bind(&Pmap_t::value_type::second, _1)));
```
// show the result
for_each(ptr_map.begin(), ptr_map.end(),
    bind(&Thing::print,
        bind(&Pmap_t::value_type::second, _1)));

// set the internal value
for_each(ptr_map.begin(), ptr_map.end(),
    bind(&Thing::set_value,
        bind(&Pmap_t::value_type::second, _1), int1));

// show the result
for_each(ptr_map.begin(), ptr_map.end(),
    bind(&Thing::print,
        bind(&Pmap_t::value_type::second, _1)));

**A handy short-cut for member functions when no binding is needed**

C++11 also includes a function template called `mem_fn` (note the spelling) which replaces the `mem_fun` and `mem_fun_ref` adapters in the C++98 Standard Library. It uses the same sophisticated template programming as `bind`, and so is “smarter” than its deprecated predecessors. This one adapter works for both containers of objects and containers of pointers. Like `bind`, `mem_fn` creates and returns a function object that can be used with function call syntax to call the wrapped function, both for a supplied object and a pointer to an object.

```cpp
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_fn` 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 supplied to the function object to play the role of “this” object:

```cpp
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 more easily than `bind`. 