How the Adapters and Binders Work

David Kieras

Prepared for EECS 381, Fall 2004

What code gets generated when we write

```
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
...
vector<int> v;
void foo(char, int);

for_each(v.begin(), v.end(), bind1st(ptr_fun(foo), 'c'));
```

The last statement, using `for_each`, is the "target statement" whose instantiation will be explained. While this explanation is limited to for_each, bind1st, and ptr_fun, the principles are general and apply to the other algorithms, adapters, and binders.

What happens in the target statement is that each `int` in the vector is supplied as the second argument to `foo`, whose first argument is always `'c'`. The process of generating the code from the involved templates is basically simple, but requires some patience to work through. Basically, there are some clever template definitions that guide the compiler to instantiate function object classes and create the appropriate objects, the last one of which can behave as a simple function object in the `for_each` instantiation. Read the following carefully, with a cup of coffee, glass of beer, etc., handy, to see Template Magic in action.

First, the Std. Lib has a template class (a struct) to hold typedefs for binary (two-argument) functions:

```
template <typename Arg1, typename Arg2, typename Result>
struct binary_function
{
	typedef Arg1   first_argument_type;
	typedef Arg2   second_argument_type;
	typedef Result result_type;
};
```

All this does is provide some "standard" typedefs so that the other templates can consistently refer to the types of the first argument, second argument, and result of a function using the same typedef names. We can inherit from a template class in a class by providing the template arguments. Thus if we instantiate

```
class Demo : public binary_function<char, int, void> {

};
```

the Demo class will end up with three public typedefs, as if we had written:

```
class Demo {
public:
	typedef char   first_argument_type;
	typedef int    second_argument_type;
	typedef void   result_type;
};
```

These typedefs can then be used in the rest of the class declaration, and because they are public, they can be used by any templates that use this template as a member or parameter.

Now, let's arrange to store a function pointer in a function object, whose operator() simply calls the function using the stored pointer. We declare the function pointer type and set up the typedefs from `binary_function`. We do this with another class template, called `pointer_to_binary_function`, declared as follows:

```
template <typename Arg1, typename Arg2, typename Result>
class pointer_to_binary_function : public binary_function<Arg1, Arg2, Result> {
public:
	explicit pointer_to_binary_function(Result (*fptr_)(Arg1, Arg2)) : fptr(fptr_) {}
	Result operator()(Arg1 x, Arg2 y) const {return fptr(x, y);}
private:
	Result (*fptr)(Arg1, Arg2);
};
```

The member variable `fptr` is our function pointer that returns `Result` type, given `Arg1` and `Arg2` type arguments. The constructor (tagged "`explicit`") simply initializes the function pointer member variable using a supplied function pointer, such as a function name. The function call operator simply calls the function using the two arguments `x` and `y`, which must be of type `Arg1` and `Arg2`.

One more template, a function template, is necessary to get us to the first instantiation in the target statement.

```
template <class Arg1, class Arg2, class Result>
inline
pointer_to_binary_function<Arg1, Arg2, Result>
ptr_fun(Result (*f)(Arg1, Arg2))
{
	return pointer_to_binary_function<Arg1, Arg2, Result>(f);
}
```

This is a function template that uses Template Magic Trick #1: *Use a function template to create and return a class template object*. The compiler will figure out what the template parameters are simply from the expression using the function template, and these can be used to easily instantiate a class template with the desired parameters.

If you call the template function `ptr_fun` with a function pointer argument (such as a function name), because the compiler knows the function declaration, it knows the return type and parameter types. So it can then instantiate the `pointer_to_binary_function` class with the corresponding template parameters, and create an object of that type, initialized with the supplied function pointer.

Let's stop and apply this much to the target expression:

```
for_each(v.begin(), v.end(), bind1st(ptr_fun(foo), 'c'));
```

`ptr_fun(foo)` results in a template instantiation for `ptr_fun` that looks something like this:

```
pointer_to_binary_function<char, int, void> // type of returned object
ptr_fun(void (*f)(char, int))
{return pointer_to_binary_function<char, int, void>(f); }
```

which when executed, creates and returns an object whose class name is
`pointer_to_binary_function<void, int, char>`
and is instantiated as:

```
class pointer_to_binary_function : public binary_function<char, int, void> {
public:
	typedef char first_argument_type;   // inherited from binary_function
	typedef int second_argument_type;
	typedef void result_type;

	explicit pointer_to_binary_function(void (*fptr_)(char, int)) :  fptr(fptr_) {}
	void operator()(char x, int y) const {return fptr(x, y);}
private:
	void (*fptr)(char, int);
};
```

where the `fptr` member has been initialized to be the address of function `foo`. Note that if we use an object of this class like a function, we will actually be calling `foo`, and still need to supply two arguments: a `char` and an `int`. So now we have a function object that we can use like a function, but we still have to supply two arguments. We want to be able to call that function with the first argument always being the same value, and the second being available to be set to whatever value the iterator points to in the `for_each`.

We'll do this with yet another function object class, in which we will save the first value, and then the function call operator will accept a value for the second argument only, and then call the function with the saved first argument and the supplied second object. This second function object is what actually gets used as the third argument in the `for_each`, playing the role of the function that gets applied to every element of the container.

This function object class is called `binder1st` - it does the "binding" so it is called a "binder". The `binder1st` is a class template that just takes an `Operation` as a template parameter. This operation will in fact be the `pointer_to_binary_function` object we have just created. Look back at that class and you will see that it has ended up with typedefs for the first and second argument types and the return type. These will be used as the argument type and returned type for the single-argument function object we will define:

```
template <typename Operation>
class binder1st {
public:
	binder1st(const Operation& x, const typename Operation::first_argument_type& y)
		: op(x), value(y) {}
	typename Operation::result_type
		operator()(const typename Operation::second_argument_type& x) const
			{return op(value, x);}
private:
	Operation op;
	typename Operation::first_argument_type value;
};
```

The constructor initializes the function object with an `Operation` object, stored as the member variable `op`, and a value, stored as the member variable `value`, to use as the argument to be the first argument every time we call the function. The type of this value is that supplied by the `first_argument_type` typedef belonging to `Operation`.

The function call operator returns a value whose type is also supplied by `Operation`, and takes a single parameter, whose type is supplied by `Operation` as the second argument type. The function call operator simply calls the `Operation` function object `op` using the stored `value`, and the single argument from `binder1st` function call operator.

The last step is to create a `binder1st` object easily, using another case of Template Magic Trick #1. This is the `bind1st` function template:

```
template <class Operation, class T>
inline
binder1st<Operation>
bind1st(const Operation& op, const T& x)
{
	return binder1st<Operation>(op, typename Operation::first_argument_type(x));
}
```

Analogous to the first Magic Trick, when `bind1st` is called with a function object `op`, it will create and return a `binder1st` object using the `Operation` type as the template parameter, initialized with `op` and an object of the first argument type initialized by the supplied value of `x`.

So, lets go back to the target statement:

```
for_each(v.begin(), v.end(), bind1st(ptr_fun(foo), 'c'));
```

Look back and see that the expression
`ptr_fun(foo)`
resulted in a `pointer_to_binary_function` object whose class name is `pointer_to_binary_function<chair, int, void>` and whose `first_argument_type` is `char`, `second_argument_type` is `int`, and `result_type` is `void`, and whose function call operator calls `foo` with the supplied arguments.

Now, the `bind1st` call is analyzed by the compiler: it sees that the first argument `op` has the type and value from the `ptr_fun`, namely a `pointer_to_binary_function<char, int, void>` object that has been initialized with the function pointer to `foo`, and the second argument has type `char`. So the compiler instantiates the `bind1st` call as something like:

```
binder1st<pointer_to_binary_function<char, int, void> > // returned type
bind1st(const pointer_to_binary_function<char, int, void>& op, const char & x)
{
return binder1st<pointer_to_binary_function<char, int, void> > (op, char(x));
}
```

This creates a `binder1st` object instantiated with the supplied function object (from `ptr_fun(foo)`) and the supplied `char` value, `'c'`. So the result of `bind1st` is an object of type `binder1st`, whose instantiation looks something like this:

```
class binder1st {
public:
binder1st(const pointer_to_binary_function<char, int, void>& x, const char& y)
		: op(x), value(y) {}
	void operator()(const int& x) const {return op(value, x);}
private:
	pointer_to_binary_function<char, int, void> op;
	char value;
};
```

The object itself is initialized with 'c' for the char value, and a function object op that simply wraps a function pointer to foo. The function call operator simply calls foo with the char value, and the int argument x.

And there it is, folks! This is a function object, that when called as a function with a single parameter x, calls a two-argument function, using a stored value for the first argument, and x for the second argument! At this point, the for_each instantiation proceeds normally, with the binder1st function object simply playing the role of the usual function pointer or function object. The for_each template

```
template<class InputIterator, class Function>
inline
Function
for_each(InputIterator first, InputIterator last, Function f)
{
	for (InputIterator it = first; it != last; ++it)
		f(*it);
	return f;
}
```

will get instantiated as:

```
inline
binder1st<pointer_to_binary_function<char, int, void> // returned type
for_each(vector<int>::iterator first, vector<int>::iterator last,
binder1st<pointer_to_binary_function<char, int, void> f)
{
	for (vector<int>::iterator it = first; it != last; ++it)
		f(*it);
	return f;
}
```

So the binder1st object gets applied to each dereferenced value in the vector.

This seems incredibly convoluted, but once the compiler has instantiated this code, it gets compiled, and then the normal optimizations will kick in. Note the "inline" declaration on the function templates, and the fact that the constructors and function call operators in the function object classes will normally be inlined as well. By the time the compiler is finished, the code that you wrote:

```
for_each(v.begin(), v.end(), bind1st(ptr_fun(foo), 'c'));
```

will in fact end up essentially as if you wrote:

```
for(vector<int>::iterator it = v.begin(); it != v.end(); ++it)
	foo('c', *it);
```

In other words, all of the template instantiation stuff is gone! It was just a massive Magic Trick to make the compiler create the code we wanted!

Note: This explanation is based on somewhat simplified declarations from the Metrowerks version of the C++ Standard Library. Other implementation may differ in details or specifics, but the overall principle is the same.