Library 9
Bind

How Does the Bind Library Improve Your Programs?

- Adapts functions and function objects for use with Standard Library algorithms
- Consistent syntax for creating binders
- Powerful functional composition

When using the algorithms from the Standard Library, you often need to supply them with a function or a function object. This is an excellent way of customizing the behavior of algorithms, but you often end up writing new function objects because you don't have the tools necessary for functional composition and adaptation of argument order or arity. Although the Standard Library does offer some productive tools, such as `bind1st` and `bind2nd`, this is rarely enough. Even when the functionality suffices, that often implies suffering from awkward syntax that obfuscates the code for programmers who are not familiar with those tools. What you need, then, is a solution that both adds functionality and normalizes the syntax for creating function objects on-the-fly, and this is what Boost.Bind does.

In effect, a generalized binder is a sort of lambda expression, because through functional composition we can more or less construct local, unnamed functions at the call site. There are many cases where this is desirable, because it serves three purposes—reducing the amount of code, making the code easier to understand, and localizing behavior, which in turn implies more effective maintenance. Note that there is another Boost library, Boost.Lambda, which takes these properties even further. Boost.Lambda is covered in the next chapter. Why shouldn't you just

skip ahead to that library? Because most of the time, Boost.Bind does everything you need when it comes to binding, and the learning curve isn't as steep.

One of the keys to the success of Bind is the uniform syntax for creating function objects and the few requirements on types that are to be used with the library. The design takes focus away from how to write the code that works with your types, and sets it to where we are all most interested—how the code works and what it actually does. When using adaptors from the Standard Library, such as `ptr_fun` and `mem_fun_ref`, code quickly becomes unnecessarily verbose because we have to provide these adaptors in order for the arguments to adhere to the requirements of the algorithms. This is not the case with Boost.Bind, which uses a much more sophisticated deduction system, and a straightforward syntax when the automatic deduction cannot be applied. The net effect of using Bind is that you'll write less code that is easier to understand.

How Does Bind Fit with the Standard Library?

Conceptually, Bind is a generalization of the existing Standard Library functions `bind1st` and `bind2nd`, with additional functionality that allows for more sophisticated functional composition. It also alleviates the need to use adaptors for pointers to functions and pointers to class members, which saves coding and potential errors. Boost.Bind also covers some of the popular extensions to the C++ Standard Library, such as the SGI extensions `compose1` and `compose2`, and also the `select1st` and `select2nd` functions. So, Bind does fit with the Standard Library, and it does so very well indeed. The need for such functionality is acknowledged, and at last in part addressed by the Standard Library, and also in popular extensions to the STL. Boost.Bind has been accepted for the upcoming Library Technical Report.

Bind

Header: `"boost/bind.hpp"`

The Bind library creates function objects that bind to a function (free function or member function). Rather than supplying all of the arguments to the function directly, arguments can be delayed, meaning that a binder can be used to create a

function object with changed arity (number of arguments) for the function it binds to, or to reorder the arguments any way you like.

The return types of the overloaded versions of the function `bind` are unspecified—that is, there is no guarantee for what the signature of a returned function object is. Sometimes, you need to store that object somewhere, rather than just passing it directly to another function—when this need arises, you want to use Boost.Function, which is covered in "Library 11: Function." The key to understanding what the `bind`-functions return is to grok the transformation that is taking place. Using one of the overloaded `bind` functions—`template<class R, class F> unspecified-1 bind(F f)`—as an example, this would be (quoting from the online documentation), "A function object λ such that the expression λ(v1, v2, ..., vm) is equivalent to f(), implicitly converted to R." Thus, the function that is bound is stored inside the binder, and the result of subsequent invocations on that function object yields the return value from the function (if any)—that is, the template parameter `R`. The implementation that we're covering here supports up to nine function arguments.

The implementation of Bind involves a number of functions and classes, but as users, we do not directly use anything other than the overloaded function `bind`. All binding takes place through the `bind` function, and we can never depend on the type of the return value. When using `bind`, the placeholders for arguments (called `_1`, `_2`, and `so on`) do not need to be introduced with a using declaration or directive, because they reside in an unnamed namespace. Thus, there is rarely a reason for writing one of the following lines when using Boost.Bind.

```
using boost::bind;
using namespace boost;
```

As was mentioned before, the current implementation of Boost.Bind supports nine placeholders (`_1`, `_2`, `_3`, and so forth), and therefore also up to nine arguments. It's instructive to at least browse through the synopsis for a high-level understanding of how the type deduction is performed, and when/why this does not always work. Parsing the signatures for member function pointers and free functions takes a while for the eye to get used to, but it's useful. You'll see that there are overloads for both free functions and class member functions. Also, there are overloads for each distinct number of arguments. Rather than listing the synopsis here, I encourage you to visit Boost.Bind's documentation at www.boost.org.

Usage

Boost.Bind offers a consistent syntax for both functions and function objects, and even for value semantics and pointer semantics. We'll start with some simple examples to get to grips with the usage of vanilla bindings, and then move on to functional composition through nested binds. One of the keys to understanding how to use `bind` is the concept of placeholders. Placeholders denote the arguments that are to be supplied to the resulting function object, and Boost.Bind supports up to nine such arguments. The placeholders are called `_1`, `_2`, `_3`, `_4`, and so on up to `_9`, and you use them in the places where you would ordinarily add the argument. As a first example, we shall define a function, `nine_arguments`, which is then called using a `bind` expression.

```
#include <iostream>
#include "boost/bind.hpp"

void nine_arguments(
  int i1,int i2,int i3,int i4,
    int i5,int i6,int i7,int i8, int i9) {
    std::cout << i1 << i2 << i3 << i4 << i5
      << i6 << i7 << i8 << i9 << '\n';
}

int main() {
  int i1=1,i2=2,i3=3,i4=4,i5=5,i6=6,i7=7,i8=8,i9=9;
  (boost::bind(&nine_arguments,_9,_2,_1,_6,_3,_8,_4,_5,_7))
    (i1,i2,i3,i4,i5,i6,i7,i8,i9);
}
```

In this example, you create an unnamed temporary binder and immediately invoke it by passing arguments to its function call operator. As you can see, the order of the placeholders is scrambled—this illustrates the reordering of arguments. Note also that placeholders can be used more than once in an expression. The output of this program is as follows.

```
921638457
```

This shows that the placeholders correspond to the argument with the placeholder's number—that is, `_1` is substituted with the first argument, `_2` with the second argument, and so on. Next, you'll see how to call member functions of a class.

Calling a Member Function

Let's take a look at calling member functions using `bind`. We'll start by doing something that also can be done with the Standard Library, in order to compare and contrast that solution with the one using Boost.Bind. When storing elements of some class type in Standard Library containers, a common need is to call a member function on some or all of these elements. This can be done in a loop, and is all-too-often implemented thusly, but there are better solutions. Consider the following simple class, `status`, which we'll use to show that the ease of use and power of Boost.Bind is indeed tremendous.

```
class status {
  std::string name_;
  bool ok_;
public:
  status(const std::string& name):name_(name),ok_(true) {}

  void break_it() {
    ok_=false;
  }

  bool is_broken() const {
    return ok_;
  }

  void report() const {
    std::cout << name_ << " is " <<
      (ok_ ? "working nominally":"terribly broken") << '\n';
  }
};
```

If we store instances of this class in a `vector`, and we need to call the member function `report`, we might be tempted to do it as follows.

```
std::vector<status> statuses;
statuses.push_back(status("status 1"));
statuses.push_back(status("status 2"));
statuses.push_back(status("status 3"));
statuses.push_back(status("status 4"));

statuses[1].break_it();
statuses[2].break_it();
```

```
for (std::vector<status>::iterator it=statuses.begin();
        it!=statuses.end();++it) {
  it->report();
}
```

This loop does the job correctly, but it's verbose, inefficient (due to the multiple calls to `statuses.end()`), and not as clear as using the algorithm from the Standard Library that exists for exactly this purpose, `for_each`. To use `for_each` to replace the loop, we need to use an adaptor for calling the member function `report` on the `vector` elements. In this case, because the elements are stored by value, what we need is the adaptor `mem_fun_ref`.

```
std::for_each(
  statuses.begin(),
  statuses.end(),
  std::mem_fun_ref(&status::report));
```

This is a correct and sound way to do it—it is quite terse, and there can be no doubt as to what the code is doing. The equivalent code for doing this using `Boost.Bind` follows.[1]

```
std::for_each(
  statuses.begin(),
  statuses.end(),
  boost::bind(&status::report,_1));
```

This version is equally clear and understandable. This is the first real use of the aforementioned placeholders of the Bind library, and what we're telling both the compiler and the reader of our code is that `_1` is to be substituted for an actual argument by the function invoking the binder. Although this code does save a few characters when typing, there is no big difference between the Standard Library `mem_fun_ref` and `bind` for this particular case, but let's reuse this example and change the container to hold pointers instead.

```
std::vector<status*> p_statuses;
p_statuses.push_back(new status("status 1"));
p_statuses.push_back(new status("status 2"));
p_statuses.push_back(new status("status 3"));
p_statuses.push_back(new status("status 4"));
```

1. It should be noted that `boost::mem_fn`, which has also been accepted for the Library Technical Report, would work just as well for the cases where there are no arguments. `mem_fn` supersedes `std::mem_fun and std::mem_fun_ref`.

```
p_statuses[1]->break_it();
p_statuses[2]->break_it();
```

We can still use both the Standard Library, but we can no longer use `mem_fun_ref`. We need help from the adaptor `mem_fun`, which is considered a bit of a misnomer, but again does the job that needs to be done.

```
std::for_each(
  p_statuses.begin(),
  p_statuses.end(),
  std::mem_fun(&status::report));
```

Although this works too, the syntax has changed, even though we are trying to do something very similar. It would be nice if the syntax was identical to the first example, so that the focus is on *what* the code really does rather than *how* it does it. Using `bind`, we do not need to be explicit about the fact that we are dealing with elements that are pointers (this is already encoded in the type of the container, and redundant information of this kind is typically unnecessary for modern libraries).

```
std::for_each(
  p_statuses.begin(),
  p_statuses.end(),
  boost::bind(&status::report,_1));
```

As you can see, this is exactly what we did in the previous example, which means that if we understood `bind` then, we should understand it now, too. Now that we have decided to switch to using pointers, we are faced with another problem, namely that of lifetime control. We must manually deallocate the elements of `p_statuses`, and that is both error prone and unnecessary. So, we may decide to start using smart pointers, and (again) change our code.

```
std::vector<boost::shared_ptr<status> > s_statuses;
s_statuses.push_back(
  boost::shared_ptr<status>(new status("status 1")));
s_statuses.push_back(
  boost::shared_ptr<status>(new status("status 2")));
s_statuses.push_back(
  boost::shared_ptr<status>(new status("status 3")));
s_statuses.push_back(
  boost::shared_ptr<status>(new status("status 4")));
```

```
s_statuses[1]->break_it();
s_statuses[2]->break_it();
```

Now, which adaptor from the Standard Library do we use? `mem_fun` and `mem_fun_ref` do not apply, because the smart pointer doesn't have a member function called `report`, and thus the following code fails to compile.

```
std::for_each(
  s_statuses.begin(),
  s_statuses.end(),
  std::mem_fun(&status::report));
```

The fact of the matter is that we lucked out—the Standard Library cannot help us with this task.[2] Thus, we have to resort to the same type of loop that we wanted to get rid of—or use Boost.Bind, which doesn't complain at all, but delivers exactly what we want.

```
std::for_each(
  s_statuses.begin(),
  s_statuses.end(),
  boost::bind(&status::report,_1));
```

Again, this example code is identical to the example before (apart from the different name of the container). The same syntax is used for binding, regardless of whether value semantics or pointer semantics apply, and even when using smart pointers. Sometimes, having a different syntax helps the understanding of the code, but in this case, it doesn't—the task at hand is to call a member function on elements of a container, nothing more and nothing less. The value of a consistent syntax should not be underestimated, because it helps both the person who is writing the code and all who later need to maintain the code (of course, we don't write code that actually needs maintenance, but for the sake of argument, let's pretend that we do).

These examples have demonstrated a very basic and common use case where Boost.Bind excels. Even though the Standard Library does offer some basic tools that do the same thing, we have seen that Bind offers both the consistency of syntax and additional functionality that the Standard Library currently lacks.

2. It will do so in the future, because both `mem_fn` and `bind` will be part of the future Standard Library.

A Look Behind the Curtain

After you start using Boost.Bind, it is inevitable; you will start to wonder how it actually works. It seems as magic when `bind` deduces the types of the arguments and return type, and what's the deal with the placeholders, anyway? We'll have a quick look on some of the mechanisms that drives such a beast. It helps to know a little about how `bind` works, especially when trying to decipher the wonderfully succinct and direct error messages the compiler emits at the slightest mistake. We will create a very simple binder that, at least in part, mimics the syntax of Boost.Bind. To avoid stretching this digression over several pages, we shall only support one type of binding, and that is for a member function taking a single argument. Moreover, we won't even get bogged down with the details of how to handle cv-qualification and its ilk; we'll just keep it simple.

First of all, we need to be able to deduce the return type, the class type, and the argument type for the function that we are to bind. We do this with a function template.

```
template <typename R, typename T, typename Arg>
  simple_bind_t<R,T,Arg> simple_bind(
    R (T::*fn)(Arg),
    const T& t,
    const placeholder&) {
      return simple_bind_t<R,T,Arg>(fn,t);
}
```

The preceding might seem a little intimidating at first, and by all rights it is because we have yet to define part of the machinery. However, the part to focus on here is where the type deduction takes place. You'll note that there are three template parameters to the function, `R`, `T`, and `Arg`. `R` is the return type, `T` is the class type, and `Arg` is the type of the (single) argument. These template parameters are what makes up the first argument to our function—that is, `R (T::*f)(Arg)`. Thus, passing a member function with a single formal parameter to `simple_bind` permits the compiler to deduce `R` as the member function's return type, `T` as the member function's class, and `Arg` as the member function's argument type. `simple_bind`'s return type is a function object that is parameterized on the same types as `simple_bind`, and whose constructor receives a pointer to the member function and an instance of the class (`T`). `simple_bind` simply ignores the placeholder (the last argument to the function), and the reason why I've included it in the first place is to simulate the syntax of Boost.Bind. In a

better implementation of this concept, we would obviously need to make use of that argument, but now we allow ourselves the luxury of letting it pass into oblivion. The implementation of the function object is fairly straightforward.

```
template <typename R,typename T, typename Arg>
  class simple_bind_t {
  typedef R (T::*fn)(Arg);
  fn fn_;
  T t_;
public:
  simple_bind_t(fn f,const T& t):fn_(f),t_(t) {}

  R operator()(Arg& a) {
    return (t_.*fn_)(a);
  }
};
```

As we saw in `simple_bind`'s implementation, the constructor accepts two arguments: the first is the pointer to a member function and the second is a reference to `const T` that is copied and later used to invoke the function with a user-supplied argument. Finally, the function call operator returns `R`, the return type of the member function, and accepts an `Arg` argument, which is the type of the argument to be passed to the member function. The somewhat obscure syntax for invoking the member function is this:

```
(t_.*fn_)(a);
```

`.*` is the pointer-to-member operator, used when the first operand is of `class T`; there's also another pointer-to-member operator, `->*`, which is used when the first operand is a pointer to `T`. What remains is to create a placeholder—that is, a variable that is used in place of the actual argument. We can create such a placeholder by using an unnamed namespace containing a variable of some type; let's call it `placeholder`:

```
namespace {
  class placeholder {};
  placeholder _1;
}
```

Let's create a simple class and a small application for testing this.

```
class Test {
public:
  void do_stuff(const std::vector<int>& v) {
    std::copy(v.begin(),v.end(),
      std::ostream_iterator<int>(std::cout," "));
  }
};

int main() {
  Test t;
  std::vector<int> vec;
  vec.push_back(42);
  simple_bind(&Test::do_stuff,t,_1)(vec);
}
```

When we instantiate the function `simple_bind` with the preceding arguments, the types are automatically deduced; `R` is `void`, `T` is `Test`, and `Arg` is a reference to `const std::vector<int>`. The function returns an instance of `simple_bind_t<void,Test,Arg>`, on which we immediately invoke the function call operator by passing the argument `vec`.

Hopefully, `simple_bind` has given you an idea of how binders work. Now, it's time to get back to Boost.Bind!

More on Placeholders and Arguments

The first example demonstrated that `bind` supports up to nine arguments, but it will serve us well to look a bit more closely at how arguments and placeholders work. First of all, it's important to note that there is an important difference between free functions and member functions—when binding to a member function, the first argument to the `bind` expression must be an instance of the member function's class! The easiest way to think about this rule is that this explicit argument substitutes the implicit `this` that is passed to all non-static member functions. The diligent reader will note that, in effect, this means that for binders to member functions, only (sic!) eight arguments are supported, because the first will be used for the actual object. The following example defines a free function `print_string` and a class `some_class` with a member function `print_string`, soon to be used in `bind` expressions.

```
#include <iostream>
#include <string>
#include "boost/bind.hpp"
```

```
class some_class {
public:
  typedef void result_type;
  void print_string(const std::string& s) const {
    std::cout << s << '\n';
  }
};

void print_string(const std::string s) {
  std::cout << s << '\n';
}

int main() {
  (boost::bind(&print_string,_1))("Hello func!");
  some_class sc;
  (boost::bind(&some_class::print_string,_1,_2))
    (sc,"Hello member!");
}
```

The first `bind` expression binds to the free function `print_string`. Because the function expects one argument, we need to use one placeholder (`_1`) to tell `bind` which of its arguments will be passed as the first argument of `print_string`. To invoke the resulting function object, we must pass the `string` argument to the function call operator. The argument is a `const std::string&`, so passing a string literal triggers invocation of `std::string`'s converting constructor.

```
(boost::bind(&print_string,_1))("Hello func!");
```

The second binder adapts a member function, `print_string` of `some_class`. The first argument to `bind` is a pointer to the member function. However, a pointer to a non-static member function isn't really a pointer.[3] We must have an object before we can invoke the function. That's why the `bind` expression must state that there are two arguments to the binder, both of which are to be supplied when invoking it.

```
boost::bind(&some_class::print_string,_1,_2);
```

3. Yes, I know how weird this sounds. It's still true, though.

To see why this makes sense, consider how the resulting function object can be used. We must pass to it both an instance of `some_class` and the argument to `print_string`.

```
(boost::bind(&some_class::print_string,_1,_2))(sc,"Hello member!");
```

The first argument to the function call operator is `this` —that is, the instance of `some_class`. Note that the first argument can be a pointer (smart or raw) or a reference to an instance; `bind` is very accommodating. The second argument to the function call operator is the member function's one argument. In this case, we've "delayed" both arguments—that is, we defined the binder such that it expects to get both the object and the member function's argument via its function call operator. We didn't have to do it that way, however. For example, we could create a binder that invokes `print_string` on the same object each time it is invoked, like so:

```
(boost::bind(&some_class::print_string,some_class(),_1))
  ("Hello member!");
```

The resulting function object already contains an instance of `some_class`, so there's only need for one placeholder (`_1`) and one argument (a string) for the function call operator. Finally, we could also have created a so-called *nullary* function object by also binding the string, like so:

```
(boost::bind(&some_class::print_string,
  some_class(),"Hello member!"))();
```

These examples clearly show the versatility of `bind`. It can be used to delay all, some, or none of the arguments required by the function it encapsulates. It can also handle reordering arguments any way you see fit; just order the placeholders according to your needs. Next, we'll see how to use `bind` to create sorting predicates on-the-fly.

Dynamic Sorting Criteria

When sorting the elements of a container, we sometimes need to create function objects that define the sorting criteria—we need to do so if we are missing relational operators, or if the existing relational operators do not define the sorting criteria we are interested in. We can sometimes use the comparison function

objects from the Standard Library (`std::greater`, `std::greater_equal`, and so forth), but only use comparisons that already exist for the types—we cannot define new ones at the call site. We'll use a class called `personal_info` for the purpose of showing how Boost.Bind can help us in this quest. `personal_info` contains the first name, last name, and age, and it doesn't provide any comparison operators. The information is immutable upon creation, and can be retrieved using the member functions `name`, `surname`, and `age`.

```
class personal_info {
  std::string name_;
  std::string surname_;
  unsigned int age_;

public:
  personal_info(
    const std::string& n,
    const std::string& s,
    unsigned int age):name_(n),surname_(s),age_(age) {}

  std::string name() const {
    return name_;
  }

  std::string surname() const {
    return surname_;
  }

  unsigned int age() const {
    return age_;
  }
};
```

We make the class *OutputStreamable* by supplying the following operator:

```
std::ostream& operator<<(
  std::ostream& os,const personal_info& pi) {
  os << pi.name() << ' ' <<
    pi.surname() << ' ' << pi.age() << '\n';
  return os;
}
```

If we are to sort a container with elements of type `personal_info`, we need to supply a sorting predicate for it. Why would we omit the relational operators

from `personal_info` in the first place? One reason is because there are several possible sorting options, and we cannot know which is appropriate for different users. Although we could also opt to provide different member functions for different sorting criteria, this would add the burden of having all relevant sorting criteria encoded in the class, which is not always possible. Fortunately, it is easy to create the predicate at the call site by using `bind`. Let's say that we need the sorting to be performed based on the age (available through the member function `age`). We could create a function object just for that purpose.

```
class personal_info_age_less_than :
  public std::binary_function<
    personal_info,personal_info,bool> {
public:
  bool operator()(
    const personal_info& p1,const personal_info& p2) {
     return p1.age()<p2.age();
  }
};
```

We've made the `personal_info_age_less_than` adaptable by publicly inheriting from `binary_function`. Deriving from `binary_function` provides the appropriate `typedef`s needed when using, for example, `std::not2`. Assuming a `vector`, `vec`, containing elements of type `personal_info`, we would use the function object like this:

```
std::sort(vec.begin(),vec.end(),personal_info_age_less_than());
```

This works fine as long as the number of different comparisons is limited. However, there is a potential problem in that the logic is defined in a different place, which can make the code harder to understand. With a long and descriptive name such as the one we've chosen here, there shouldn't be a problem, but not all cases are so clear-cut, and there is a real chance that we'd need to supply a slew of function objects for greater than, less than or equal to, and so on.

So, how can Boost.Bind help? Actually, it helps us out three times for this example. If we examine the problem at hand, we find that there are three things we need to do, the first being to bind a logical operation, such as `std::less`. This is easy, and gives us the first part of the code.

```
boost::bind<bool>(std::less<unsigned int>(),_1,_2);
```

Note that we are explicitly adding the return type by supplying the `bool` parameter to `bind`. This is sometimes necessary, both on broken compilers and in contexts where the return type cannot be deduced. If a function object contains a `typedef`, `result_type`, there is no need to explicitly name the return type.[4] Now, we have a function object that accepts two arguments, both of type `unsigned int`, but we can't use it just yet, because the elements have the type `personal_info`, and we need to extract the age from those elements and pass the age as arguments to `std::less`. Again, we can use `bind` to do that.

```
boost::bind(
  std::less<unsigned int>(),
  boost::bind(&personal_info::age,_1),
  boost::bind(&personal_info::age,_2));
```

Here, we create two more binders. The first one calls `personal_info::age` with the main binder's function call operator's first argument (`_1`). The second one calls `personal_info::age` with the main binder's function call operator's second argument (`_2`). Because `std::sort` passes two `personal_info` objects to the main binder's function call operator, the result is to invoke `personal_info::age` on each of two `personal_info` objects from the `vector` being sorted. Finally, the main binder passes the ages returned by the two new, inner binders' function call operator to `std::less`. This is exactly what we need! The result of invoking this function object is the result of `std::less`, which means that we have a valid comparison function object easily used to sort a container of `personal_info` objects. Here's how it looks in action:

```
std::vector<personal_info> vec;
vec.push_back(personal_info("Little","John",30));
vec.push_back(personal_info("Friar", "Tuck",50));
vec.push_back(personal_info("Robin", "Hood",40));

std::sort(
  vec.begin(),
  vec.end(),
  boost::bind(
    std::less<unsigned int>(),
    boost::bind(&personal_info::age,_1),
    boost::bind(&personal_info::age,_2)));
```

4. The Standard Library function objects all have `result_type` defined, so they work with `bind`'s return type deduction mechanism.

We could sort differently simply by binding to another member (variable or function) from `personal_info`—for example, the last name.

```
std::sort(
  vec.begin(),
  vec.end(),
  boost::bind(
    std::less<std::string>(),
    boost::bind(&personal_info::surname,_1),
    boost::bind(&personal_info::surname,_2)));
```

This is a great technique, because it offers an important property: simple functionality implemented at the call site. It makes the code easy to understand and maintain. Although it is technically possible to sort using binders based upon complex criteria, it is not wise. Adding more logic to the `bind` expressions quickly loses clarity and succinctness. Although it is sometimes tempting to do more in terms of binding, strive to write binders that are as clever as the people who must maintain it, but no more so.

Functional Composition, Part I

One problem that's often looking for a solution is to compose a function object out of other functions or function objects. Suppose that you need to test an `int` to see whether it is greater than 5 and less than, or equal to, 10. Using "regular" code, you would do something like this:

```
if (i>5 && i<=10) {
  // Do something
}
```

When processing elements of a container, the preceding code only works if you put it in a separate function. When this is not desirable, using a nested `bind` can express the same thing (note that this is typically not possible using `bind1st` and `bind2nd` from the Standard Library). If we decompose the problem, we find that we need operations for *logical and* (`std::logical_and`), *greater than* (`std::greater`), and *less than or equal to* (`std::less_equal`). The *logical and* should look something like this:

```
boost::bind(std::logical_and<bool>(),_1,_2);
```

Then, we need another predicate that answers whether `_1` is *less than* or *equal to* 10.

```
boost::bind(std::greater<int>(),_1,5);
```

Then, we need another predicate that answers whether `_1` is *less than* or *equal to* 10.

```
boost::bind(std::less_equal<int>(),_1,10);
```

Finally, we need to logically *and* those two together, like so:

```
boost::bind(
  std::logical_and<bool>(),
  boost::bind(std::greater<int>(),_1,5),
  boost::bind(std::less_equal<int>(),_1,10));
```

A nested `bind` such as this is relatively easy to understand, though it has postfix order. Still, one can almost read the code literally and determine the intent. Let's put this binder to the test in an example.

```
std::vector<int> ints;

ints.push_back(7);
ints.push_back(4);
ints.push_back(12);
ints.push_back(10);

int count=std::count_if(
  ints.begin(),
  ints.end(),
  boost::bind(
    std::logical_and<bool>(),
    boost::bind(std::greater<int>(),_1,5),
    boost::bind(std::less_equal<int>(),_1,10)));

std::cout << count << '\n';

std::vector<int>::iterator int_it=std::find_if(
  ints.begin(),
  ints.end(),
  boost::bind(std::logical_and<bool>(),
    boost::bind(std::greater<int>(),_1,5),
```

```
    boost::bind(std::less_equal<int>(),_1,10)));

if (int_it!=ints.end()) {
  std::cout << *int_it << '\n';
}
```

It is important to carefully indent the code properly when using nested `bind`s, because the code can quickly become hard to understand if one neglects sensible indentation. Consider the preceding clear code, and then look at the following obfuscated example.

```
std::vector<int>::iterator int_it=
  std::find_if(ints.begin(),ints.end(),
    boost::bind<bool>(
    std::logical_and<bool>(),
    boost::bind<bool>(std::greater<int>(),_1,5),
      boost::bind<bool>(std::less_equal<int>(),_1,10)));
```

This is a general problem with long lines, of course, but it becomes apparent when using constructs such as those described here, where long statements are the rule rather than the exception. So, please be nice to your fellow programmers by making sure that your lines wrap in a way that makes them easy to read.

One of the hard-working reviewers for this book asked why, in the previous example, two equivalent binders were created, and if it wouldn't make more sense to create a binder object and use it two times. The answer is that because we can't know the exact type of the binder (it's implementation defined) that's created when we call `bind`, we have no way of declaring a variable for it. Also, the type typically is very complex, because its signature includes all of the type information that's been captured (and deduced automatically) in the function `bind`. However, it is possible to store the resulting function objects using other facilities—for example, those from Boost.Function. See "Library 11: Function" for details on how this is accomplished.

The composition outlined here corresponds to a popular extension to the Standard Library, namely the function `compose2` from the SGI STL, also known as `compose_f_gx_hx` in the (now deprecated) Boost.Compose library.

Functional Composition, Part II

Another useful functional composition is known as `compose1` in SGI STL, and `compose_f_gx` in Boost.Compose. These functionals offer a way to call two

functions with an argument, and have the result of the innermost function passed to the first function. An example sometimes says more than a thousand contrived words, so consider the scenario where you need to perform two arithmetic operations on container elements of floating point type. We first add 10% to the values, and then reduce the values with 10%; the example could also serve as a useful lesson to quite a few people working in the financial sector.

```
std::list<double> values;
values.push_back(10.0);
values.push_back(100.0);
values.push_back(1000.0);

std::transform(
  values.begin(),
  values.end(),
  values.begin(),
  boost::bind(
 std::multiplies<double>(),0.90,
   boost::bind<double>(
      std::multiplies<double>(),_1,1.10)));

std::copy(
  values.begin(),
  values.end(),
  std::ostream_iterator<double>(std::cout," "));
```

How do you know which of the nested `bind`s will be called first? As you've probably already noticed, it is always the innermost `bind` that is evaluated first. This means that we could write the equivalent code somewhat differently.

```
std::transform(
  values.begin(),
  values.end(),
  values.begin(),
  boost::bind<double>(
  std::multiplies<double>(),
    boost::bind<double>(
      std::multiplies<double>(),_1,1.10),0.90));
```

Here, we change the order of the arguments passed to the `bind`, tacking on the argument to the first `bind` last in the expression. Although I do not recommend this practice, it is useful for understanding how arguments are passed to `bind` functions.

Value or Pointer Semantics in *bind* Expressions?

When we pass an instance of some type to a `bind` expression, it is copied, unless we explicitly tell `bind` not to copy it. Depending on what we are doing, this can be of vital importance. To see what goes on behind our backs, we will create a `tracer` class that will tell us when it is default constructed, copy constructed, assigned to, and destructed. That way, we can easily see how different uses of `bind` affect the instances that we pass. Here is the `tracer` class in its entirety.

```
class tracer {
public:
  tracer() {
  std::cout << "tracer::tracer()\n";
}

tracer(const tracer& other) {
  std::cout << "tracer::tracer(const tracer& other)\n";
}

tracer& operator=(const tracer& other) {
  std::cout <<
    "tracer& tracer::operator=(const tracer& other)\n";
  return *this;
}

~tracer() {
  std::cout << "tracer::~tracer()\n";
}

void print(const std::string& s) const {
  std::cout << s << '\n';
  }
};
```

We put our `tracer` class to work with a regular `bind` expression like the one that follows.

```
tracer t;
boost::bind(&tracer::print,t,_1)
  (std::string("I'm called on a copy of t\n"));
```

Running this code produces the following output, which clearly shows that there is copying involved.

```
tracer::tracer()
tracer::tracer(const tracer& other)
tracer::tracer(const tracer& other)
tracer::tracer(const tracer& other)
tracer::~tracer()
tracer::tracer(const tracer& other)
tracer::~tracer()
tracer::~tracer()
I'm called on a copy of t

tracer::~tracer()
```

If we had been using objects where copying was expensive, we probably could not afford to use `bind` this way. There is an advantage to the copying, however. It means that the `bind` expression and its resulting binder are not dependent on the lifetime of the original object (`t` in this case), which is often the exact behavior desired. To avoid the copies, we must tell `bind` that we intend to pass it a reference that it is supposed to use rather than a value. We do this with `boost::ref` and `boost::cref` (for reference and reference to `const`, respectively), which are also part of the Boost.Bind library. Using `boost::ref` with our `tracer` class, the testing code now looks like this:

```
tracer t;
boost::bind(&tracer::print,boost::ref(t),_1)(
  std::string("I'm called directly on t\n"));
```

Executing the code gives us this:

```
tracer::tracer()
I'm called directly on t
tracer::~tracer
```

That's exactly what's needed to avoid unnecessary copying. The `bind` expression uses the original instance, which means that there are no copies of the `tracer` object. Of course, it also means that the binder is now dependent upon the lifetime of the `tracer` instance. There's also another way of avoiding copies; just pass the argument by pointer rather than by value.

```
tracer t;
boost::bind(&tracer::print,&t,_1)(
  std::string("I'm called directly on t\n"));
```

So, `bind` always copies. If you pass by value, the object is copied and that may be detrimental on performance or cause unwanted effects. To avoid copying the object, you can either use `boost::ref`/`boost::cref` or use pointer semantics.

Virtual Functions Can Also Be Bound

So far, we've seen how `bind` can work with non-member functions and non-virtual member functions, but it is, of course, also possible to bind a virtual member function. With Boost.Bind, you use virtual functions as you would non-virtual functions—that is, just bind to the virtual function in the base class that first declared the member function virtual. That makes the binder useful with all derived types. If you bind against a more derived type, you restrict the classes with which the binder can be used.[5] Consider the following classes named `base` and `derived`:

```
class base {
public:
  virtual void print() const {
    std::cout << "I am base.\n";
  }
  virtual ~base() {}
};

class derived : public base {
public:
  void print() const {
    std::cout << "I am derived.\n";
  }
};
```

Using these classes, we can test the binding of a virtual function like so:

```
derived d;
base b;
boost::bind(&base::print,_1)(b);
boost::bind(&base::print,_1)(d);
```

5. This is no different than when declaring a pointer to a class in order to invoke a virtual member function. The more derived the type pointed to, the fewer classes can be bound to the pointer.

Running this code clearly shows that this works as one would hope and expect.

```
I am base.
I am derived.
```

The fact that virtual functions are supported should come as no surprise to you, but now we've shown that it works just like other functions. On a related note, what would happen if you `bind` a member function that is later redefined by a derived class, or a virtual function that is public in the base class but private in the derived? Will things still work? If so, which behavior would you expect? Well, the behavior does not change whether you are using Boost.Bind or not. Thus, if you `bind` to a function that is redefined in another class—that is, it's not virtual and the derived class adds a member function with an identical signature—the version in the base class is called. If a function is hidden, the binder can still be invoked, because it explicitly accesses the function in the base class, which works even for hidden member functions. Finally, if the virtual function is declared public in the base class, but is private in a derived class, invoking the function on an instance of the derived class succeeds, because the access is through a base instance, where the member is public. Of course, such a case indicates a seriously flawed design.

Binding to Member Variables

There are many occasions when you need to `bind` data members rather than member functions. For example, when using `std::map` or `std::multimap`, the element type is `std::pair<key const,data>`, but the information you want to use is often not the key, but the data. Suppose you want to pass the data from each element in a `map` to a function that takes a single argument of the data type. You need to create a binder that forwards the `second` member of each element (of type `std::pair`) to the bound function. Here's code that illustrates how to do that:

```
void print_string(const std::string& s) {
  std::cout << s << '\n';
}

std::map<int,std::string> my_map;
my_map[0]="Boost";
my_map[1]="Bind";
```

```
std::for_each(
  my_map.begin(),
  my_map.end(),
    boost::bind(&print_string, boost::bind(
      &std::map<int,std::string>::value_type::second,_1)));
```

You can `bind` to a member variable just as you can with a member function, or a free function. It should be noted that to make the code easier to read (and write), it's a good idea to use short and convenient names. In the previous example, the use of a `typedef` for the `std::map` helps improve readability.

```
typedef std::map<int,std::string> map_type;
boost::bind(&map_type::value_type::second,_1)));
```

Although the need to `bind` to member variables does not arise as often as for member functions, it is still very convenient to be able to do so. Users of SGI STL (and derivatives thereof) are probably familiar with the `select1st` and `select2nd` functions. They are used to select the `first` or the `second` member of `std::pair`, which is the same thing that we're doing in this example. Note that `bind` works with arbitrary types and arbitrary names, which is definitively a plus.

To Bind or Not to Bind

The great flexibility brought by the Boost.Bind library also offers a challenge for the programmer, because it is sometimes very tempting to use a binder, although a separate function object is warranted. Many tasks can and should be accomplished with the help of Bind, but it's an error to go too far—and the line is drawn where the code becomes hard to read, understand, and maintain. Unfortunately, the position of the line greatly depends on the programmers that share (by reading, maintaining, and extending) the code, as their experience must dictate what is acceptable and what is not. The advantage of using specialized function objects is that they can typically be made quite self-explanatory, and to provide the same clear message using binders is a challenge that we must diligently try to overcome. For example, if you need to create a nested `bind` that you have trouble understanding, chances are that you have gone too far. Let me explain this with code.

```
#include <iostream>
#include <string>
```

```
#include <map>
#include <vector>
#include <algorithm>
#include "boost/bind.hpp"

void print(std::ostream* os,int i) {
  (*os) << i << '\n';
}

int main() {
  std::map<std::string,std::vector<int> > m;
  m["Strange?"].push_back(1);
  m["Strange?"].push_back(2);
  m["Strange?"].push_back(3);
  m["Weird?"].push_back(4);
  m["Weird?"].push_back(5);

  std::for_each(m.begin(),m.end(),
    boost::bind(&print,&std::cout,
    boost::bind(&std::vector<int>::size,
    boost::bind(
      &std::map<std::string,
        std::vector<int> >::value_type::second,_1))));
}
```

What does the preceding code actually do? There are people who read code like this fluently,[6] but for many of us mortals, it takes some time to figure out what's going on. Yes, the binder calls the member function `size` on whatever exists as the `pair` member `second` (the `std::map<std::string,std::vector <int> >::value_type`). In cases like this, where the problem is simple yet complex to express using a binder, it often makes sense to create a small function object instead of a complex binder that some people will definitely have a hard time understanding. A simple function object that does the same thing could look something like this:

```
class print_size {
  std::ostream& os_;
  typedef std::map<std::string,std::vector<int> > map_type;
public:
  print_size(std::ostream& os):os_(os) {}
```

6. Hello, Peter Dimov.

```
  void operator()(
    const map_type::value_type& x) const {
      os_ << x.second.size() << '\n';
  }
};
```

The great advantage in this case comes when we are using the function object, whose name is self-explanatory.

```
std::for_each(m.begin(),m.end(),print_size(std::cout));
```

This (the source for the function object and the actual invocation) is to be compared with the version using a binder.

```
std::for_each(m.begin(),m.end(),
  boost::bind(&print,&std::cout,
  boost::bind(&std::vector<int>::size,
  boost::bind(
    &std::map<std::string,
      std::vector<int> >::value_type::second,_1))));
```

Or, if we had been a bit more responsible and created terse `typedef`s for the `vector` and `map`:

```
std::for_each(m.begin(),m.end(),
  boost::bind(&print,&std::cout,
  boost::bind(&vec_type::size,
  boost::bind(&map_type::value_type::second,_1))));
```

That's a bit easier to parse, but it's still a bit too much.

Although there may be some good arguments for using the `bind` version, I think that the point is clear—binders are incredibly useful tools that should be used responsibly, where they add value. This is very, very common when using the Standard Library containers and algorithms. But when things get too complicated, do it the old fashioned way.

Let Binders Handle State

There are several options available to use when creating a function object like `print_size`. The version that we created in the previous section stored a reference to a `std::ostream`, and used that `ostream` to print the return value of `size`

for the member `second` on the `map_type::value_type` argument. Here's the original `print_size` again:

```
class print_size {
  std::ostream& os_;
  typedef std::map<std::string,std::vector<int> > map_type;
public:
   print_size(std::ostream& os):os_(os) {}

  void operator()(
    const map_type::value_type& x) const {
      os_ << x.second.size() << '\n';
  }
};
```

An important observation for this class is that is has state, through the stored `std::ostream`. We could remove the state by adding the `ostream` as an argument to the function call operator. This would mean that the function object becomes stateless.

```
class print_size {
  typedef std::map<std::string,std::vector<int> > map_type;
public:
  typedef void result_type;
  result_type operator()(std::ostream& os,
    const map_type::value_type& x) const {
      os << x.second.size() << '\n';
  }
};
```

Note that this version of `print_size` is well behaved when used with `bind`, through the addition of the `result_type typedef`. This relieves users from having to explicitly state the return type of the function object when using `bind`. In this new version of `print_size`, users need to pass an `ostream` as an argument when invoking it. That's easy when using binders. Rewriting the example from the previous section with the new `print_size` gives us this:

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
```

```
#include "boost/bind.hpp"

// Definition of print_size omitted

int main() {
  typedef std::map<std::string,std::vector<int> > map_type;
  map_type m;
  m["Strange?"].push_back(1);
  m["Strange?"].push_back(2);
  m["Strange?"].push_back(3);
  m["Weird?"].push_back(4);
  m["Weird?"].push_back(5);

  std::for_each(m.begin(),m.end(),
    boost::bind(print_size(),boost::ref(std::cout),_1));
}
```

The diligent reader might wonder why `print_size` isn't a free function now, because it doesn't carry state anymore. In fact, it can be

```
void print_size(std::ostream& os,
  const std::map<std::string,std::vector<int> >::value_type& x) {
  os << x.second.size() << '\n';
}
```

But there are more generalizations to consider. Our current version of `print_size` requires that the second argument to the function call operator be a reference to `const std::map<std::string,std::vector<int> >`, which isn't very general. We can do better, by parameterizing the function call operator on the type. This makes `print_size` usable with any argument that contains a public member called `second`, which in turn has a member function `size`. Here's the improved version:

```
class print_size {
public:
  typedef void result_type;
  template <typename Pair> result_type operator()
    (std::ostream& os,const Pair& x) const {
    os << x.second.size() << '\n';
  }
};
```

Usage is the same with this version as the previous, but it's much more flexible. This kind of generalization becomes more important than usual when creating function objects that can be used in `bind` expressions. Because the number of cases where the function objects can be used increase markedly, most any potential generalization is worthwhile. In that vein, there is one more change that we could make to further relax the requirements for types to be used with `print_size`. The current version of `print_size` requires that the second argument of the function call operator be a pair-like object—that is, an object with a member called `second`. If we decide to require only that the argument contain a member function `size`, the function object starts to really deserve its name.

```
class print_size {
public:
  typedef void result_type;
  template <typename T> void operator()
    (std::ostream& os,const T& x) const {
    os << x.size() << '\n';
  }
};
```

Of course, although `print_size` is now true to its name, we require more of the user for the use case that we've already considered. Usage now includes "manually" binding to `map_type::value_type::second`.

```
std::for_each(m.begin(),m.end(),
  boost::bind(print_size(),boost::ref(std::cout),
    boost::bind(&map_type::value_type::second,_1)));
```

Such are often the tradeoffs when using `bind`—generalizations can only take you so far before starting to interfere with usability. Had we taken things to an extreme, and removed even the requirement that there be a member function `size`, we'd complete the circle and be back where we started, with a `bind` expression that's just too complex for most programmers.

```
std::for_each(m.begin(),m.end(),
  boost::bind(&print[7],&std::cout,
  boost::bind(&vec_type::size,
  boost::bind(&map_type::value_type::second,_1))));
```

7. The `print` function would obviously be required, too, without some lambda facility.

A Boost.Bind and Boost.Function Teaser

Although the material that we have covered in this chapter shouldn't leave you wanting for more, there is actually a very useful synergy between Boost.Bind and another library, Boost.Function, that provides still more functionality. We shall see more of the added value in "Library 11: Function," but I'd like to give you a hint of what's to come. As we've seen, there is no apparent way of storing our binders for later use—we only know that they are compatible function objects with some (unknown) signature. But, when using Boost.Function, storing functions for later invocation is exactly what the library does, and thanks to the compatibility with Boost.Bind, it's possible to assign binders to functions, saving them for later invocation. This is an enormously useful concept, which enables adaptation and promotes loose coupling.

Bind Summary

Use Bind when

- You need to bind a call to a free function, and some or all of its arguments
- You need to bind a call to a member function, and some or all of its arguments
- You need to compose nested function objects

The existence of a generalized binder is a tremendously useful tool when it comes to writing terse, coherent code. It reduces the number of small function objects created for adapting functions/function objects, and combinations of functions. Although the Standard Library already offers a small part of the functionality found in Boost.Bind, there are significant improvements that make Boost.Bind the better choice in most places. In addition to the simplification of existing features, Bind also offers powerful functional composition features, which provide the programmer with great power without negative effects on maintenance. If you've taken the time to learn about `bind1st`, `bind2nd`, `ptr_fun`, `mem_fun_ref`, and so forth, you'll have little or no trouble transitioning to Boost.Bind. If you've yet to start using the current binder offerings from the C++ Standard Library, I strongly suggest that you start by using Bind, because it is both easier to learn and more powerful.

I know many programmers who have yet to experience the wonders of binders in general, and function composition in particular. If you used to be one of them, I'm hoping that this chapter has managed to convey some of the tremendous power that is brought forth by the concept as such. Moreover, think about the implications this type of function, declared and defined at the call site, will have on maintenance. It's going to be a breeze compared to the dispersion of code that can easily be caused by small, innocent-looking[8] function objects that are scattered around the classes merely to provide the correct signature and perform a trivial task.

The Boost.Bind library is created and maintained by Peter Dimov, who has, besides making it such a complete facility for binding and function composition, also managed to make it work cleanly for most compilers.

8. But they're not.