
10.2. Predefined Function Objects and Binders

As mentioned in Section 6.10.2, page 239, the C++ standard library provides many predefined function objects and binders that allow you to compose them into more sophisticated function objects. This ability, called *functional composition*, requires fundamental function objects and adapters, which are both presented here. To use these function objects and binders, you must include the header file `<functional>`:

```
#include <functional>
```

10.2.1. Predefined Function Objects

Table 10.1 lists all predefined function objects (`bit_and`, `bit_or`, and `bit_xor` are available since C++11).

Table 10.1. Predefined Function Objects

Expression	Effect
`negate<`*type*`>()`	`-` *param*
`plus<`*type*`>()`	*param1* `+` *param2*
`minus<`*type*`>()`	*param1* `-` *param2*
`multiplies<`*type*`>()`	*param1* `*` *param2*
`divides<`*type*`>()`	*param1* `/` *param2*
`modulus<`*type*`>()`	*param1* `%` *param2*
`equal_to<`*type*`>()`	*param1* `==` *param2*
`not_equal_to<`*type*`>()`	*param1* `!=` *param2*
`less<`*type*`>()`	*param1* `<` *param2*
`greater<`*type*`>()`	*param1* `>` *param2*
`less_equal<`*type*`>()`	*param1* `<=` *param2*
`greater_equal<`*type*`>()`	*param1* `>=` *param2*
`logical_not<`*type*`>()`	`!` *param*
`logical_and<`*type*`>()`	*param1* `&&` *param2*
`logical_or<`*type*`>()`	*param1* `||` *param2*
`bit_and<`*type*`>()`	*param1* `&` *param2*
`bit_or<`*type*`>()`	*param1* `|` *param2*
`bit_xor<`*type*`>()`	*param1* `^` *param2*

`less<>` is the default criterion whenever objects are sorted or compared by sorting functions and associative containers. Thus, default sorting operations always produce an ascending order (*element* `<` *nextElement*). `equal_to<>` is the default equivalence criterion for unordered containers.

To compare internationalized strings, the C++ standard library provides the ability to use locale objects as function objects so that they can be used as a sorting criterion for strings (see Section 16.3, page 868, for details).

10.2.2. Function Adapters and Binders

A function adapter is a function object that enables the composition of function objects with each other, with certain values, or with special functions (according to the composite pattern in [*GoF:DesignPatterns*]). However, over time, the way function objects are composed changed. In fact, all such features that were provided for C++98 are deprecated since C++11, which introduced more convenient and more flexible adapters. Here, I first present the current way to compose function objects. In Section 10.2.4, page 497, I give a very brief overview of the deprecated features.

Table 10.2 lists the function adapters provided by the C++ standard library since C++11.

Table 10.2. Predefined Function Adapters

Expression	Effect
`bind(`*op*`,`*args*`...)`	Binds *args* to *op*
`mem_fn(`*op*`)`	Calls *op*`()` as a member function for an object or pointer to object
`not1(`*op*`)`	Unary negation: `!`*op*`(`*param*`)`
`not2(`*op*`)`	Binary negation: `!`*op*`(`*param1*`,`*param2*`)`

The most important adapter is `bind()` . It allows you to:

- Adapt and compose new function objects out of existing or predefined function objects
- Call global functions
- Call member functions for objects, pointers to objects, and smart pointers to objects

The `bind()` Adapter

In general, `bind()` binds parameters for *callable objects* (see Section 4.4, page 54). Thus, if a function, member function, function object, or lambda requires some parameters, you can bind them to specific or passed arguments. Specific arguments you simply name. For passed arguments, you can use the predefined *placeholders* `_1` , `_2` , ... defined in namespace `std::placeholders` .

A typical application of binders is to specify parameters when using the predefined function objects provided by the C++ standard library (see Section 10.2.1, page 486). For example:

Click here to view code image

```
// fo/bind1.cpp

#include <functional>
#include <iostream>

int main()
{
    auto plus10 = std::bind(std::plus<int>(),
                            std::placeholders::_1,
                            10);
    std::cout << "+10:    " << plus10(7) << std::endl;

    auto plus10times2 = std::bind(std::multiplies<int>(),
                                  std::bind(std::plus<int>(),
                                            std::placeholders::_1,
                                            10),
                                  2);
    std::cout << "+10 *2: " << plus10times2(7) << std::endl;

    auto pow3 = std::bind(std::multiplies<int>(),
                          std::bind(std::multiplies<int>(),
                                    std::placeholders::_1,
                                    std::placeholders::_1),
                          std::placeholders::_1);
    std::cout << "x*x*x:  " << pow3(7) << std::endl;

    auto inversDivide = std::bind(std::divides<double>(),
                                  std::placeholders::_2,
                                  std::placeholders::_1);
    std::cout << "invdiv: " << inversDivide(49,7) << std::endl;
}
```

Here, four different binders that represent function objects are defined. For example, `plus10` , defined as

```
std::bind(std::plus<int>(),
          std::placeholders::_1,
          10)
```

represents a function object, which internally calls `plus<>` (i.e., operator `+`), with a placeholder `_1` as first parameter/operand and `10` as second parameter/operand. The placeholder `_1` represents the first argument passed to the expression as a whole. Thus, for any argument passed to this expression, this function object yields the value of that argument `+10` .

To avoid the tedious repetition of the namespace `placeholders` , you can use a corresponding using directive. Thus, with two using directives, you condense the whole statement:

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

bind (plus<int>(), _1, 10)    // param1+10
```

The binder can also be called directly. For example,

```
std::cout << std::bind(std::plus<int>(),_1,10)(32) << std::endl;
```

will write `42` to standard output and, if you pass this function object to an algorithm, the algorithm can apply it to every element the algorithms operates with. For example:

[Click here to view code image]

```
// add 10 to each element
std::transform (coll.begin(), coll.end(),                  // source
                coll.begin(),                              // destination
                std::bind(std::plus<int>(),_1,10));        // operation
```

In the same way, you can define a binder that represents a search criterion. For example, to find the first element that is greater than `42`, you bind `greater<>` with the passed argument as first and `42` as second operator:

[Click here to view code image]

```
// find first element >42
auto pos = std::find_if (coll.begin(),coll.end(),
                         std::bind(std::greater<int>(),_1,42))
```

Note that you always have to specify the argument type of the predefined function object used. If the type doesn't match, a type conversion is forced, or the expression results in a compile-time error.

The remaining statements in this example program demonstrate that you can nest binders to compose even more complicated function objects. For example, the following expression defines a function object that adds `10` to the passed argument and then multiplies it by `2` (namespaces omitted):

```
bind(multiplies<int>(),                  // (param1+10)*2
         bind(plus<int>(),_1,
                    10),
         2);
```

As you can see, the expressions are evaluated from the inside to the outside.

Similarly, we can raise a value to the power of `3` by combining two `multiplies<>` objects with three placeholders for the argument passed:

[Click here to view code image]

```
bind(multiplies<int>(),                  // (param1*param1)*param1
         bind(multiplies<int>(),_1,
                                _1),
         _1);
```

The final expression defines a function object, where the arguments for a division are swapped. Thus, it divides the second argument by the first argument:

```
bind(divides<double>(),_2,              // param2/param1
                         _1);
```

Thus, the example program as a whole has the following output:

```
+10:    17
+10 *2: 34
x*x*x:  343
invdiv: 0.142857
```

Section 6.10.3, page 241, offers some other examples of the use of binders. Section 10.3.1, page 499, provides the same functionality using lambdas.

Calling Global Functions

The following example demonstrates how `bind()` can be used to call global functions (see Section 10.3.3, page 502, for a version with lambdas):

[Click here to view code image]

```
// fo/compose3.cpp

#include <iostream>
#include <algorithm>
#include <functional>
#include <locale>
#include <string>
using namespace std;
using namespace std::placeholders;
```

```
char myToupper (char c)
{
    std::locale loc;
    return std::use_facet<std::ctype<char> >(loc).toupper(c);
}

int main()
{
    string s("Internationalization");
    string sub("Nation");

    // search substring case insensitive
    string::iterator pos;
    pos = search (s.begin(),s.end(),           // string to search in
                  sub.begin(),sub.end(),       // substring to search
                  bind(equal_to<char>(),       // compar. criterion
                       bind(myToupper,_1),
                       bind(myToupper,_2)));
    if (pos != s.end()) {
        cout << "\"" << sub << "\" is part of \"" << s << "\""
             << endl;
    }
}
```

Here, we use the `search()` algorithm to check whether `sub` is a substring in `s`, when case sensitivity doesn't matter. With

```
bind(equal_to<char>(),
     bind(myToupper,_1),
     bind(myToupper,_2)));
```

we create a function object calling:

`myToupper(`*param1*`)==myToupper(`*param2*`)`

where `myToupper()` is our own convenience function to convert the characters of the strings into uppercase (see Section 16.4.4, page 891, for details).

Note that `bind()` internally copies passed arguments. To let the function object use a reference to a passed argument, use `ref()` or `cref()` (see Section 5.4.3, page 132). For example:

```
void incr (int& i)
{
    ++i;
}

int i=0;
bind(incr,i)();          // increments a copy of i, no effect for i
bind(incr,ref(i))();     // increments i
```

Calling Member Functions

The following program demonstrates how `bind()` can be used to call member functions (see Section 10.3.3, page 503, for a version with lambdas):

Click here to view code image

```
// fo/bind2.cpp

#include <functional>
#include <algorithm>
#include <vector>
#include <iostream>
#include <string>
using namespace std;
using namespace std::placeholders;

class Person {
  private:
    string name;
  public:
    Person (const string& n) : name(n) {
    }
    void print () const {
        cout << name << endl;
    }
    void print2 (const string& prefix) const {
        cout << prefix << name << endl;
    }
```

```
    ...
};

int main()
{
    vector<Person> coll
            = { Person("Tick"), Person("Trick"), Person("Track") };

    // call member function print() for each person
    for_each (coll.begin(), coll.end(),
              bind(&Person::print,_1));
    cout << endl;

    // call member function print2() with additional argument for each person
    for_each (coll.begin(), coll.end(),
              bind(&Person::print2,_1,"Person: "));
    cout << endl;

    // call print2() for temporary Person
    bind(&Person::print2,_1,"This is: ")(Person("nico"));
}
```

Here,

```
bind(&Person::print,_1)
```

defines a function object that calls *param1* `.print()` for a passed `Person`. That is, because the first argument is a member function, the next argument defines the object for which this member function gets called.

Any additional argument is passed to the member function. That means:

```
bind(&Person::print2,_1,"Person: ")
```

defines a function object that calls *param1* `.print2("Person: ")` for any passed `Person`.

Here, the passed objects are the members of `coll`, but in principle, you can pass objects directly. For example:

```
Person n("nico");
bind(&Person::print2,_1,"This is: ")(n);
```

calls `n.print2("This is: ")`.

The output of the program is as follows:

```
Tick
Trick
Track

Person: Tick
Person: Trick
Person: Track

This is: nico
```

Note that you can also pass pointers to objects and even smart pointers to `bind()`:

```
std::vector<Person*> cp;
...
std::for_each (cp.begin(), cp.end(),
               std::bind(&Person::print,
                         std::placeholders::_1));

std::vector<std::shared_ptr<Person>> sp;
...
std::for_each (sp.begin(), sp.end(),
               std::bind(&Person::print,
                         std::placeholders::_1));
```

Note that you can also call modifying member functions:

Click here to view code image

```
class Person {
  public:
    ...
    void setName (const std::string& n) {
        name = n;
    }
```

```
};

std::vector<Person> coll;
...
std::for_each (coll.begin(), coll.end(),          // give all Persons same name
               std::bind(&Person::setName,
                         std::placeholders::_1,
                         "Paul"));
```

Calling virtual member functions also works. If a method of the base class is bound and the object is of a derived class, the correct virtual function of the derived class gets called.

The `mem_fn()` Adapter

For member functions, you can also use the `mem_fn()` adapter, whereby you can skip the placeholder for the object the member function is called for:

```
std::for_each (coll.begin(), coll.end(),
               std::mem_fn(&Person::print));
```

Thus, for the object returned by `mem_fn(...)`, operator () simply calls the member function it was initialized with. The function call is performed for the object passed as first argument while additional arguments are passed as parameters to this member function:

[Click here to view code image]

```
std::mem_fn(&Person::print)(n);                 // calls n.print()
std::mem_fn(&Person::print2)(n,"Person: ");     // calls n.print2("Person: ")
```

However, to bind an additional argument to the function object, you again have to use `bind()`:

```
std::for_each (coll.begin(), coll.end(),
               std::bind(std::mem_fn(&Person::print2),
               std::placeholders::_1,
               "Person: "));
```

Binding to Data Members

You can also bind to data members. Consider the following example (namespaces omitted):[4]

[4] This example is based on code taken from [Karlsson:Boost], page 260, with friendly permission by the author.

[Click here to view code image]

```
map<string,int> coll;     // map of int values associated to strings
...
// accumulate all values (member second of the elements)
int sum
 = accumulate (coll.begin(), coll.end(),
               0,
               bind(plus<int>(),
                    _1,
                    bind(&map<string,int>::value_type::second,
                         _2)));
```

Here, `accumulate()` is called, which uses a binary predicate to sum up all values of all elements (see Section 11.11.1, page 623). However, because we use a map, where the elements are key/value pairs, to gain access to an element's value

```
bind(&map<string,int>::value_type::second,_2)
```

binds the passed second argument of each call of the predicate to its member `second`.

Adapters `not1()` and `not2()`

The adapters `not1()` and `not2()` can be considered as almost deprecated.[5] The only way to use them is to negate the meaning of predefined function objects. For example:

[5] In fact, they were close to being deprecated with C++11, see [N3198:DeprAdapt]

```
std::sort (coll.begin(), coll.end(),
           std::not2(std::less<int>()));
```

This looks more convenient than:

[Click here to view code image]

```
std::sort (coll.begin(), coll.end(),
           std::bind(std::logical_not<bool>(),
```

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

However, there is no real real-world scenario for `not1()` and `not2()` because you can simply use another predefined function object here:

```
std::sort (coll.begin(), coll.end(),
           std::greater_equal<int>());
```

More important, note that calling `not2()` with `less<>` is wrong anyway. You probably meant to change the sorting from ascending to descending. But the negation of `<` is `>=`, not `>`. In fact, `greater_equal<>` even leads to undefined behavior because `sort()` requires a *strict weak ordering*, which `<` provides, but `>=` does not provide because it violates the requirement to be antisymmetric (see Section 7.7, page 314). Thus, you either pass

```
greater<int>()
```

or swap the order of arguments by passing

```
bind(less<int>(),_2,_1)
```

See Section 10.2.4, page 497, for other examples using `not1()` and `not2()` with deprecated function adapters.

10.2.3. User-Defined Function Objects for Function Adapters

You can also use binders for your user-defined function objects. The following example shows a complete definition for a function object that processes the first argument raised to the power of the second argument:

```
// fo/fopow.hpp

#include <cmath>

template <typename T1, typename T2>
struct fopow
{
    T1 operator() (T1 base, T2 exp) const {
        return std::pow(base,exp);
    }
};
```

Note that the first argument and the return value have the same type, `T1`, whereas the exponent may have a different type `T2`.

The following program shows how to use the user-defined function object `fopow<>()`. In particular, it uses `fopow<>()` with the `bind()` function adapters:

Click here to view code image

```
// fo/fopow1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <functional>
#include "fopow.hpp"
using namespace std;
using namespace std::placeholders;

int main()
{
    vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // print 3 raised to the power of all elements
    transform (coll.begin(), coll.end(),                    // source
               ostream_iterator<float>(cout," "),           // destination
               bind(fopow<float,int>(),3,_1));              // operation
    cout << endl;

    // print all elements raised to the power of 3
    transform (coll.begin(), coll.end(),                    // source
               ostream_iterator<float>(cout," "),           // destination
               bind(fopow<float,int>(),_1,3));              // operation
    cout << endl;
}
```

The program has the following output:

```
3 9 27 81 243 729 2187 6561 19683
1 8 27 64 125 216 343 512 729
```

Note that `fopow<>()` is realized for types `float` and `int`. If you use `int` for both base and exponent, you'd call `pow()` with two arguments of type `int`, but this isn't portable, because according to the standard, `pow()` is overloaded for more than one but not all fundamental types:

```
transform (coll.begin(), coll.end(),
           ostream_iterator<int>(cout," "),
           bind1st(fopow<int,int>(),3));     // ERROR: ambiguous
```

See Section 17.3, page 942, for details about this problem.

10.2.4. Deprecated Function Adapters

Table 10.3 lists the predefined function adapter classes that were provided by the C++ standard library before C++11 and are deprecated now.[6] Just in case you encounter the deprecated stuff, here are some brief examples of how to use them.

[6] Although `not1()` and `not2()` are not officially deprecated, you need the other deprecated function adapters for real-world usage.

Table 10.3. Deprecated Predefined Function Adapters

Expression	Effect
`bind1st(`*op*`,`*arg*`)`	Calls *op*`(`*arg*`,`*param*`)`
`bind2nd(`*op*`,`*arg*`)`	Calls *op*`(`*param*`,`*arg*`)`
`ptr_fun(`*op*`)`	Calls `*`*op*`(`*param*`)` or `*`*op*`(`*param1*`,`*param2*`)`
`mem_fun(`*op*`)`	Calls *op*`()` as a member function for a pointer to an object
`mem_fun_ref(`*op*`)`	Calls *op*`()` as a member function for an object
`not1(`*op*`)`	Unary negation: `!`*op*`(`*param*`)`
`not2(`*op*`)`	Binary negation: `!`*op*`(`*param1*`,`*param2*`)`

Note that these adapters required certain type definitions in the functions objects used. To define these types, the C++ standard library provides special base classes for function adapters: `std::unary_function<>` and `std::binary_function<>`. These classes also are deprecated now.

Both `bind1st()` and `bind2nd()` operate like `bind()`, with fixed positions that a parameter is bound to. For example:

Click here to view code image

```
// find first element >42
std::find_if (coll.begin(),coll.end(),                  // range
              std::bind2nd(std::greater<int>(),42))     // criterion
```

However, `bind1st()` and `bind2nd()` can't be used to compose binders out of binders or pass ordinary functions directly.

`not1()` and `not2()` are "almost deprecated" because they are useful only with the other deprecated function adapters. For example:

Click here to view code image

```
std::find_if (coll.begin(), coll.end(),
              std::not1(std::bind2nd(std::modulus<int>(),2)));
```

finds the position of the first even `int` (`%2` yields `0` for even values, which `not1()` negates into `true`). However, this looks more convenient than using the new binders:

```
std::find_if (coll.begin(), coll.end(),
              std::bind(std::logical_not<bool>(),
                        std::bind(std::modulus<int>(),
                                  std::placeholders::_1,
                                  2)));
```

Being able to use a lambda is really an improvement here:

```
std::find_if (coll.begin(), coll.end(),
              [](int elem){
                  return elem%2==0;
              });
```

`ptr_fun()` was provided to be able to call ordinary functions. For example, suppose that you have a global function, which checks something for each parameter:

```
bool check(int elem);
```

To find the first element, for which the check does not succeed, you could call the following statement:

Click here to view code image

```
std::find_if (coll.begin(), coll.end(),          // range
              std::not1(std::ptr_fun(check)));   // search criterion
```

The second form of `ptr_fun()` was used when you had a global function for two parameters and, for example, you wanted to use it as a unary function:

```
// find first string that is not empty
std::find_if (coll.begin(), coll.end(),
              std::bind2nd(std::ptr_fun(std::strcmp),""));
```

Here, the `strcmp()` C function is used to compare each element with the empty C-string. If both strings match, `strcmp()` returns `0`, which is equivalent to `false`. So, this call of `find_if()` returns the position of the first element that is not the empty string.

Both `mem_fun()` and `mem_fun_ref()` were provided to define function objects that call member functions.[7] For example:

[7] These member function adapters use the auxiliary classes `mem_fun_t`, `mem_fun_ref_t`, `const_mem_fun_t`, `const_mem_fun_ref_t`, `mem_fun1_t`, `mem_fun1_ref_t`, `const_mem_fun1_t`, and `const_mem_fun1_ref_t`.

```
class Person {
  public:
    void print () const;
    ...
};

const std::vector<Person> coll;
...
// call member function print() for each person
std::for_each (coll.begin(), coll.end(),
               std::mem_fun_ref(&Person::print));
```

Note that the member functions called by `mem_fun_ref()` and `mem_fun()` and passed as arguments to `bind1st()` or `bind2nd()` must be *constant* member functions.