
6.10. Function Objects

Functional arguments for algorithms don't have to be functions. As seen with lambdas, functional arguments can be objects that behave like functions. Such an object is called a *function object*, or *functor*.[13] Instead of using a lambda, you can define a function object as an object of a class that provides a *function call operator*. This was possible even before C++11.

[13] Since C++11, the standard uses the term *function object* for every object that can be used to perform a function call. Thus, function pointers, objects of classes with operator `()` or with a conversion to a pointer to function, and lambdas are function objects. Here in this book, however, I use the term for objects of classes with operator `()` defined.

6.10.1. Definition of Function Objects

Function objects are another example of the power of generic programming and the concept of pure abstraction. You could say that anything that *behaves* like a function *is* a function. So, if you define an object that behaves as a function, it can be used like a function.

So, what is the behavior of a function? A functional behavior is something that you can call by using parentheses and passing arguments. For example:

```
function(arg1,arg2);      // a function call
```

If you want objects to behave this way, you have to make it possible to "call" them by using parentheses and passing arguments. Yes, that's possible (rarely are things not possible in C++). All you have to do is define operator `()` with the appropriate parameter types:

```
class X {
  public:
    // define "function call" operator:
    return-value operator() (arguments) const;
    ...
};
```

Now you can use objects of this class to behave like a function that you can call:

Click here to view code image

```
X fo;
...
fo(arg1,arg2);            // call operator () for function object fo
```

The call is equivalent to:

```
fo.operator()(arg1,arg2); // call operator () for function object fo
```

The following is a complete example. This is the function object version of a previous example (see Section 6.8.1, page 224) that did the same with an ordinary function:

```
// stl/foreach2.cpp

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// simple function object that prints the passed argument
class PrintInt {
  public:
    void operator() (int elem) const {
        cout << elem << ' ';
    }
};

int main()
{
    vector<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }
```

```
    // print all elements
    for_each (coll.cbegin(), coll.cend(),      // range
              PrintInt());                     // operation
    cout << endl;
}
```

The class `PrintInt` defines objects for which you can call operator `()` with an `int` argument. The expression

```
PrintInt()
```

in the statement

```
for_each (coll.cbegin(), coll.cend(),
          PrintInt());
```

creates a temporary object of this class, which is passed to the `for_each()` algorithm as an argument. The `for_each()` algorithm is written like this:

Click here to view code image

```
namespace std {
    template <typename Iterator, typename Operation>
    Operation for_each (Iterator act, Iterator end, Operation op)
    {
        while (act != end) {    // as long as not reached the end
            op(*act);           // - call op() for actual element
            ++act;              // - move iterator to the next element
        }
        return op;
    }
}
```

`for_each()` uses the temporary function object `op` to call `op(*act)` for each element `act`. If `op` is an ordinary function, `for_each()` simply calls it with `*act` as an argument. If `op` is a function object, `for_each()` calls its operator `()` with `*act` as an argument. Thus, in this example program, `for_each()` calls:

```
PrintInt::operator()(*act)
```

You may be wondering what all this is good for. You might even think that function objects look strange, nasty, or nonsensical. It is true that they do complicate code. However, function objects are more than functions, and they have some advantages:

1. **Function objects are "functions with state."** Objects that behave like pointers are smart pointers. This is similarly true for objects that behave like functions: They can be "smart functions" because they may have abilities beyond operator `()`. Function objects may have other member functions and attributes. This means that function objects have a state. In fact, the same functionality, represented by two different function objects of the same type, may have different states at the same time. This is not possible for ordinary functions. Another advantage of function objects is that you can initialize them at runtime before you use/call them.
2. **Each function object has its own type.** Ordinary functions have different types only when their signatures differ. However, function objects can have different types even when their signatures are the same. In fact, each functional behavior defined by a function object has its own type. This is a significant improvement for generic programming using templates because you can pass functional behavior as a template parameter. Doing so enables containers of different types to use the same kind of function object as a sorting criterion, ensuring that you don't assign, combine, or compare collections that have different sorting criteria. You can even design hierarchies of function objects so that you can, for example, have different, special kinds of one general criterion.
3. **Function objects are usually faster than ordinary functions.** The concept of templates usually allows better optimization because more details are defined at compile time. Thus, passing function objects instead of ordinary functions often results in better performance.

In the rest of this subsection, I present some examples that demonstrate how function objects can be "smarter" than ordinary functions. Chapter 10, which deals only with function objects, provides more examples and details. In particular, Chapter 10 shows how to benefit from the ability to pass functional behavior as a template parameter.

Suppose that you want to add a certain value to all elements of a collection. If you know the value you want to add at compile time, you could use an ordinary function:

```
void add10 (int& elem)
{
    elem += 10;
}

void f1()
{
    vector<int> coll;
    ...

    for_each (coll.begin(), coll.end(),     // range
              add10);                       // operation
}
```

If you need different values that are known at compile time, you could use a template instead:

```
template <int theValue>
void add (int& elem)
{
    elem += theValue;
}

void f1()
{
    vector<int> coll;
    ...

    for_each (coll.begin(), coll.end(),     // range
              add<10>);                     // operation
}
```

If you process the value to add at runtime, things get complicated. You must pass that value to the function before the function is called. This normally results in a global variable that is used both by the function that calls the algorithm and by the function that is called by the algorithm to add that value. This is messy style.

If you need such a function twice, with two different values to add, and if both values are processed at runtime, you can't achieve this with one ordinary function. You must either pass a tag or write two different functions. Did you ever copy the definition of a function because it had a static variable to keep its state and you needed the same function with another state at the same time? This is exactly the same type of problem.

With function objects, you can write a "smarter" function that behaves in the desired way. Because it may have a state, the object can be initialized by the correct value. Here is a complete example:[14]

[14] The auxiliary function `PRINT_ELEMENTS()` was introduced in Section 6.6, page 216.

Click here to view code image

```
// stl/add1.cpp

#include <list>
#include <algorithm>
#include <iostream>
#include "print.hpp"
using namespace std;

// function object that adds the value with which it is initialized
class AddValue {
  private:
    int theValue;    // the value to add
  public:
    // constructor initializes the value to add
    AddValue(int v) : theValue(v) {
    }

    // the "function call" for the element adds the value
    void operator() (int& elem) const {
        elem += theValue;
    }
};

int main()
{
    list<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    PRINT_ELEMENTS(coll,"initialized:                ");

    // add value 10 to each element
    for_each (coll.begin(), coll.end(),     // range
              AddValue(10));                // operation

    PRINT_ELEMENTS(coll,"after adding 10:            ");

    // add value of first element to each element
    for_each (coll.begin(), coll.end(),     // range
              AddValue(*coll.begin()));     // operation

    PRINT_ELEMENTS(coll,"after adding first element: ");
}
```

After the initialization, the collection contains the values 1 to 9 :

```
initialized:            1 2 3 4 5 6 7 8 9
```

The first call of for_each() adds 10 to each value:

```
for_each (coll.begin(), coll.end(),      // range
          AddValue(10));                 // operation
```

Here, the expression AddValue(10) creates an object of type AddValue that is initialized with the value 10 . The constructor of AddValue stores this value as the member theValue . Inside for_each() , " () " is called for each element of coll . Again, this is a call of operator () for the passed temporary function object of type AddValue . The actual element is passed as an argument. The function object adds its value 10 to each element. The elements then have the following values:

```
after adding 10:            11 12 13 14 15 16 17 18 19
```

The second call of for_each() uses the same functionality to add the value of the first element to each element. This call initializes a temporary function object of type AddValue with the first element of the collection:

```
AddValue(*coll.begin())
```

The output is then as follows:

```
after adding first element: 22 23 24 25 26 27 28 29 30
```

See Section 11.4, page 520, for a modified version of this example, in which the AddValue function object type is a template for the type of value to add.

With this technique, two different function objects can solve the problem of having a function with two states at the same time. For example, you could simply declare two function objects and use them independently:

Click here to view code image

```
AddValue addx(x);      // function object that adds value x
AddValue addy(y);      // function object that adds value y

for_each (coll.begin(),coll.end(),       // add value x to each element
          addx);
...
for_each (coll.begin(),coll.end(),       // add value y to each element
          addy);
...
for_each (coll.begin(),coll.end(),       // add value x to each element
          addx);
```

Similarly, you can provide additional member functions to query or to change the state of the function object during its lifetime. See Section 10.1.3, page 482, for a good example.

Note that for some algorithms, the C++ standard library does not specify how often function objects are called for each element, and it might happen that different copies of the function object are passed to the elements. This might have some nasty consequences if you use function objects as predicates. Section 10.1.4, page 483, covers this issue.

6.10.2. Predefined Function Objects

The C++ standard library contains several predefined function objects that cover fundamental operations. By using them, you don't have to write your own function objects in several cases. A typical example is a function object used as a sorting criterion. The default sorting criterion for operator < is the predefined sorting criterion less<> . Thus, if you declare

```
set<int> coll;
```

it is expanded to

```
set<int,less<int>> coll;  // sort elements with <
```

From there, it is easy to sort elements in the opposite order:

```
set<int,greater<int>> coll;  // sort elements with >
```

Another place to apply predefined function objects are algorithms. Consider the following example:

```
// stl/fo1.cpp
```

```
#include <deque>
#include <algorithm>
#include <functional>
#include <iostream>
#include "print.hpp"
using namespace std;

int main()
{
    deque<int> coll = { 1, 2, 3, 5, 7, 11, 13, 17, 19 };

    PRINT_ELEMENTS(coll,"initialized: ");

    // negate all values in coll
    transform (coll.cbegin(),coll.cend(),       // source
               coll.begin(),                    // destination
               negate<int>());                  // operation
    PRINT_ELEMENTS(coll,"negated:     ");

    // square all values in coll
    transform (coll.cbegin(),coll.cend(),       // first source
               coll.cbegin(),                   // second source
               coll.begin(),                    // destination
               multiplies<int>());              // operation
    PRINT_ELEMENTS(coll,"squared:     ");
}
```

First, the header for the predefined function objects is included: `<functional>`

```
#include <functional>
```

Then, two predefined function objects are used to negate and square the elements in `coll`. In

```
transform (coll.cbegin(), coll.cend(),     // source
           coll.begin(),                   // destination
           negate<int>());                 // operation
```

the expression

```
negate<int>()
```

creates a function object of the predefined class template `negate<>` that simply returns the negated element of type `int` for which it is called. The `transform()` algorithm uses that operation to transform all elements of the first collection into the second collection. If source and destination are equal, as in this case, the returned negated elements overwrite themselves. Thus, the statement negates each element in the collection.

Similarly, the function object `multiplies` is used to square all elements in `coll`:

```
transform (coll.cbegin(), coll.cend(),     // first source
           coll.cbegin(),                  // second source
           coll.begin(),                   // destination
           multiplies<int>());             // operation
```

Here, another form of the `transform()` algorithm combines elements of two collections by using the specified operation and writes the resulting elements into the third collection. Again, all collections are the same, so each element gets multiplied by itself, and the result overwrites the old value.

Thus, the program has the following output:

```
initialized: 1 2 3 5 7 11 13 17 19
negated:     -1 -2 -3 -5 -7 -11 -13 -17 -19
squared:     1 4 9 25 49 121 169 289 361
```

6.10.3. Binders

You can use special *function adapters*, or so-called *binders*, to combine predefined function objects with other values or use special cases. Here is a complete example:

Click here to view code image

```
// stl/bind1.cpp

#include <set>
#include <deque>
#include <algorithm>
#include <iterator>
```

```
#include <functional>
#include <iostream>
#include "print.hpp"
using namespace std;
using namespace std::placeholders;

int main()
{
    set<int,greater<int>> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    deque<int> coll2;

    // Note: due to the sorting criterion greater<>() elements have reverse order:
    PRINT_ELEMENTS(coll1,"initialized: ");

    // transform all elements into coll2 by multiplying them with 10
    transform (coll1.cbegin(),coll1.cend(),          // source
               back_inserter(coll2),                 // destination
               bind(multiplies<int>(),_1,10));       // operation
    PRINT_ELEMENTS(coll2,"transformed: ");

    // replace value equal to 70 with 42
    replace_if (coll2.begin(),coll2.end(),       // range
                bind(equal_to<int>(),_1,70),     // replace criterion
                42);                             // new value
    PRINT_ELEMENTS(coll2,"replaced:    ");

    // remove all elements with values between 50 and 80
    coll2.erase(remove_if(coll2.begin(),coll2.end(),
                          bind(logical_and<bool>(),
                               bind(greater_equal<int>(),_1,50),
                               bind(less_equal<int>(),_1,80))),
                coll2.end());
    PRINT_ELEMENTS(coll2,"removed:    ");
}
```

Here, the statement

Click here to view code image

```
transform (coll1.cbegin(),coll1.cend(),          // source
           back_inserter(coll2),                 // destination
           bind(multiplies<int>(),_1,10));       // operation
```

transforms all elements of `coll1` into `coll2` (inserting) while multiplying each element by `10`. To define the corresponding operation, `bind()` is used, which allows you to compose high-level function objects out of low-level function objects and placeholders, which are numeric identifiers that start with an underscore. By specifying

```
bind(multiplies<int>(),_1,10)
```

you define a function object that multiplies a first passed argument with `10`.

You could also use such a function object to multiply any value by `10`. For example, the following statements write `990` to the standard output:

```
auto f = bind(multiplies<int>(),_1,10);
cout << f(99) << endl;
```

This function object is passed to `transform()`, which is expecting as its fourth argument an operation that takes one argument; namely, the actual element. As a consequence, `transform()` calls "'multiply by `10`" for each actual element and inserts the result into `coll2`, which means that afterward `coll2` contains all values of `coll1` multiplied by `10`.

Similarly, in

Click here to view code image

```
replace_if (coll2.begin(),coll2.end(),       // range
            bind(equal_to<int>(),_1,70),     // replace criterion
            42);                             // new value
```

the following expression is used as a criterion to specify the elements that are replaced by `42`:

```
bind(equal_to<int>(),_1,70)
```

Here, `bind()` calls the binary predicate `equal_to` with the passed first parameter as first argument and `70` as second argument. Thus, the function object specified via `bind()` yields `true` if a passed argument (element of `coll2`) is equal

to `70`. As a result, the whole statement replaces each value equal to `70` by `42`.

The last example uses a combination of binders, where

```
bind(logical_and<bool>(),
     bind(greater_equal<int>(),_1,50),
     bind(less_equal<int>(),_1,80))
```

specifies for a parameter `x` the unary predicate "`x>=50&&x<=80`." This example demonstrates that you can use nested `bind()` expressions to describe even more complicated predicates and function objects. In this case, `remove_if()` uses the function object to remove all values between `50` and `80` out of the collection. In fact, `remove_if()` changes only the order and returns the new end, whereas `coll2.erase()` deletes the "removed" elements out of `coll2` (see Section 6.7.1, page 218, for details).

The output of the whole program is as follows:

```
initialized: 9 8 7 6 5 4 3 2 1
transformed: 90 80 70 60 50 40 30 20 10
replaced:    90 80 42 60 50 40 30 20 10
removed:     90 42 40 30 20 10
```

Note that the placeholders have their own namespace: `std::placeholders`. For this reason, a corresponding using directive is placed at the beginning to be able to write `_1` or `_2` for a first or second parameter of binders. Without any using directive, the last combination of binders would have to be specified as follows:

```
std::bind(std::logical_and<bool>(),
     std::bind(std::greater_equal<int>(),std::placeholders::_1,50),
     std::bind(std::less_equal<int>(),std::placeholders::_1,80))
```

This kind of programming results in *functional composition*. What is interesting is that all these function objects are usually declared inline. Thus, you use a function-like notation or abstraction, but you get good performance.

There are other ways to define function objects. For example, to call a member function for each element of a collection, you can specify the following:

```
for_each (coll.cbegin(), coll.cend(),     // range
          bind(&Person::save,_1));        // operation: Person::save(elem)
```

The function object `bind` binds a specified member function to call it for every element, which is passed here with placeholder `_1`. Thus, for each element of the collection `coll`, the member function `save()` of class `Person` is called. Of course, this works only if the elements have type `Person` or a type derived from `Person`.

Section 10.2, page 486, lists and discusses in more detail all predefined function objects, function adapters, and aspects of functional composition. Also given there is an explanation of how you can write your own function objects.

Before TR1, there were other binders and adapters for functional composition, such as `bind1st()`, `bind2nd()`, `ptr_fun()`, `mem_fun()`, and `mem_fun_ref()`, but they have been deprecated with C++11. See Section 10.2.4, page 497, for details.

6.10.4. Function Objects and Binders versus Lambdas

Lambdas are a kind of implicitly defined function object. Thus, as written in Section 6.9, page 230, lambdas usually provide the more intuitive approach to defining functional behavior of STL algorithms. In addition, lambdas should be as fast as function objects.

However, there are also some drawbacks to lambdas:

- You can't have a hidden internal state of such a function object. Instead, all data that defines a state is defined by the caller and passed as a capture.
- The advantage of specifying the functional behavior where it is needed partially goes away when it is needed at multiple places. You can define a lambda and assign it to an `auto` object then (see Section 6.9, page 232), but whether this is more readable than directly defining a function object is probably a matter of taste.