
10.1. The Concept of Function Objects

A function object, or *functor*, is an object that has operator `()` defined so that in the following example

```
FunctionObjectType fo;
...
fo(...);
```

the expression `fo()` is a call of operator `()` for the function object `fo` instead of a call of the function `fo()`. At first, you could consider a function object as an ordinary function that is written in a more complicated way. Instead of writing all the function statements inside the function body:

```
void fo() {
     statements
}
```

you write them inside the body of operator `()` of the function object class:

```
class FunctionObjectType {
  public:
    void operator() () {
         statements
    }
};
```

This kind of definition is more complicated but has three important advantages:

1. A function object might be smarter because it may have a state (associated members that influence the behavior of the function object). In fact, you can have two instances of the same function object class, which may have different states at the same time. This is not possible for ordinary functions.
2. Each function object has its own type. Thus, you can pass the type of a function object to a template to specify a certain behavior, and you have the advantage that container types with different function objects differ.
3. A function object is usually faster than a function pointer.

See Section 6.10.1, page 235, for more details about these advantages and an example that shows how function objects can be smarter than ordinary functions.

In the next two subsections, I present two other examples that go into more detail about function objects. The first example demonstrates how to benefit from the fact that each function object usually has its own type. The second example demonstrates how to benefit from the state of function objects and leads to an interesting property of the `for_each()` algorithm, which is covered in Section 10.1.3, page 482.

10.1.1. Function Objects as Sorting Criteria

Programmers often need a sorted collection of elements that have a special class (for example, a collection of `Person`s). However, you either don't want to or can't use the usual operator `<` to sort the objects. Instead, you sort the objects according to a special sorting criterion based on some member function. In this regard, a function object can help. Consider the following example:

Click here to view code image

```
// fo/sort1.cpp

#include <iostream>
#include <string>
#include <set>
#include <algorithm>
using namespace std;

class Person {
  public:
    string firstname() const;
    string lastname() const;
    ...
};

// class for function predicate
// - operator () returns whether a person is less than another person
class PersonSortCriterion {
  public:
    bool operator() (const Person& p1, const Person& p2) const {
```

```
        // a person is less than another person
        // - if the last name is less
        // - if the last name is equal and the first name is less
        return p1.lastname()<p2.lastname() ||
               (p1.lastname()==p2.lastname() &&
                p1.firstname()<p2.firstname());
    }
};

int main()
{
    // create a set with special sorting criterion
    set<Person,PersonSortCriterion> coll;
    ...

    // do something with the elements
    for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
        ...
    }
    ...
}
```

The set `coll` uses the special sorting criterion `PersonSortCriterion`, which is defined as a function object class. `PersonSortCriterion` defines operator `()` in such a way that it compares two `Person`s according to their last names and, if they are equal, to their first names. The constructor of `coll` creates an instance of class `PersonSortCriterion` automatically so that the elements are sorted according to this sorting criterion.

Note that the sorting criterion `PersonSortCriterion` is a *type*. Thus, you can use it as a template argument for the set. This would not be possible if you implement the sorting criterion as a plain function (as was done in Section 6.8.2, page 228).

All sets with this sorting criterion have their own type. You can't combine or assign a set that has a "normal" or another user-defined sorting criterion. Thus, you can't compromise the automatic sorting of the set by any operation; however, you can design function objects that represent different sorting criteria with the same type (see Section 7.8.6, page 351).

10.1.2. Function Objects with Internal State

The following example shows how function objects can be used to behave as a function that may have more than one state at the same time:

Click here to view code image

```
// fo/sequence1.cpp

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

class IntSequence {
  private:
    int value;
  public:
    IntSequence (int initialValue)      // constructor
     : value(initialValue) {
    }

    int operator() () {                 // "function call"
        return value++;
    }
};

int main()
{
    list<int> coll;

    // insert values from 1 to 9
    generate_n (back_inserter(coll),    // start
                9,                      // number of elements
                IntSequence(1));        // generates values, starting with 1

    PRINT_ELEMENTS(coll);

    // replace second to last element but one with values starting at 42
    generate (next(coll.begin()),       // start
              prev(coll.end()),         // end
              IntSequence(42));         // generates values, starting with 42
```

```
    PRINT_ELEMENTS(coll);
}
```

In this example, the function object `IntSequence` generates a sequence of integral values. Each time operator `()` is called, it returns its actual value and increments it. You can pass the start value as a constructor argument.

Two such function objects are then used by the `generate()` and `generate_n()` algorithms, which use generated values to write them into a collection: The expression

```
IntSequence(1)
```

in the statement

```
generate_n (back_inserter(coll),
            9,
            IntSequence(1));
```

creates such a function object initialized with `1`. The `generate_n()` algorithm uses it nine times to write an element, so it generates values `1` to `9`. Similarly, the expression

```
IntSequence(42)
```

generates a sequence beginning with value `42`. The `generate()` algorithm replaces the elements beginning with the second up to the last but one.[1] The output of the program is as follows:

[1] `std::next()` and `std::prev()` are provided since C++11 (see Section 9.3.2, page 443). Note that using `++coll.begin()` and `--coll.end()` might not always compile (see Section 9.2.6, page 440).

```
1 2 3 4 5 6 7 8 9
1 42 43 44 45 46 47 48 9
```

Using other versions of operator `()`, you can easily produce more complicated sequences.

By default, function objects are passed by value rather than by reference. Thus, the algorithm does not change the state of the function object. For example, the following code generates the sequence starting with value `1` twice:

```
IntSequence seq(1);    // integral sequence starting with value 1

// insert sequence beginning with 1
generate_n (back_inserter(coll), 9, seq);

// insert sequence beginning with 1 again
generate_n (back_inserter(coll), 9, seq);
```

Passing function objects by value instead of by reference has the advantage that you can pass constant and temporary expressions. Otherwise, passing `IntSequence(1)` would not be possible.

The disadvantage of passing the function object by value is that you can't benefit from modifications of the state of the function objects. Algorithms can modify the state of the function objects, but you can't access and process their final states, because they make internal copies of the function objects. However, access to the final state might be necessary, so the question is how to get a "result" from an algorithm.

There are three ways to get a "result" or "feedback" from function objects passed to algorithms:

1. You can keep the state externally and let the function object refer to it.
2. You can pass the function objects by reference.
3. You can use the return value of the `for_each()` algorithm.

The last option is discussed in the next subsection.

To pass a function object by reference, you simply have to qualify the call of the algorithm so that the function object type is a reference.[2] For example:

[2] Thanks to Philip Köster for pointing this out.

Click here to view code image

```
// fo/sequence2.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>
#include "print.hpp"
```

```
using namespace std;

class IntSequence {
  private:
    int value;
  public:
    // constructor
    IntSequence (int initialValue)
     : value(initialValue) {
    }

    // "function call"
    int operator() () {
        return value++;
    }
};

int main()
{
    list<int> coll;
    IntSequence seq(1);    // integral sequence starting with 1

    // insert values from 1 to 4
    // - pass function object by reference
    //   so that it will continue with 5
    generate_n<back_insert_iterator<list<int>>,
               int, IntSequence&>(back_inserter(coll),    // start
                                  4,       // number of elements

                                  seq);    // generates values
    PRINT_ELEMENTS(coll);

    // insert values from 42 to 45
    generate_n (back_inserter(coll),       // start
                4,                         // number of elements
                IntSequence(42));          // generates values
    PRINT_ELEMENTS(coll);

    // continue with first sequence
    // - pass function object by value
    //   so that it will continue with 5 again
    generate_n (back_inserter(coll),       // start
                4,                         // number of elements
                seq);                      // generates values
    PRINT_ELEMENTS(coll);

    // continue with first sequence again
    generate_n (back_inserter(coll),       // start
                4,                         // number of elements
                seq);                      // generates values
    PRINT_ELEMENTS(coll);
}
```

The program has the following output:

```
1 2 3 4
1 2 3 4 42 43 44 45
1 2 3 4 42 43 44 45 5 6 7 8
1 2 3 4 42 43 44 45 5 6 7 8 5 6 7 8
```

In the first call of `generate_n()`, the function object `seq` is passed by reference. To do this, the template arguments are qualified explicitly:

Click here to view code image

```
generate_n<back_insert_iterator<list<int>>,
           int, IntSequence&>(back_inserter(coll),    // start
                              4,                      // number of elements
                              seq);                   // generates values
```

As a result, the internal value of `seq` is modified after the call, and the second use of `seq` by the third call of `generate_n()` continues the sequence of the first call. However, this call passes `seq` by value:

Click here to view code image

```
generate_n (back_inserter(coll),       // start
            4,                         // number of elements
```

```
seq);                                   // generates values
```

Thus, the call does not change the state of `seq`. As a result, the last call of `generate_n()` continues the sequence with value `5` again.

10.1.3. The Return Value of `for_each()`

The effort involved with passing a function object by reference in order to access its final state is not necessary if you use the `for_each()` algorithm. `for_each()` has the unique ability to return its function object (no other algorithm can do this). Thus, you can query the state of your function object by checking the return value of `for_each()`.

The following program is a nice example of the use of the return value of `for_each()`. It shows how to process the mean value of a sequence:

Click here to view code image

```
// fo/foreach3.cpp

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

// function object to process the mean value
class MeanValue {
  private:
    long num;      // number of elements
    long sum;      // sum of all element values
  public:
    // constructor
    MeanValue () : num(0), sum(0) {
    }

    // "function call"
    // - process one more element of the sequence
    void operator() (int elem) {
        ++num;           // increment count
        sum += elem;     // add value
    }

    // return mean value
    double value () {
        return static_cast<double>(sum) / static_cast<double>(num);
    }
};

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

    // process and print mean value
    MeanValue mv = for_each (coll.begin(), coll.end(),   // range
                             MeanValue());               // operation
    cout << "mean value: " << mv.value() << endl;
}
```

The expression

```
MeanValue()
```

creates a function object that counts the number of elements and processes the sum of all element values. By passing the function object to `for_each()`, it is called for each element of the container `coll`:

```
MeanValue mv = for_each (coll.begin(), coll.end(),
                         MeanValue());
```

The function object is returned and assigned to `mv`, so you can query its state after the statement by calling: `mv.value()`. Therefore, the program has the following output:

```
mean value: 4.5
```

You could even make the class `MeanValue` a bit smarter by defining an automatic type conversion to `double`. You could then use the mean value that is processed by `for_each()` directly. See Section 11.4, page 522, for such an example.

Note that lambdas provide a more convenient way to specify this behavior (see Section 10.3.2, page 500, for a corresponding example).

However, that does not mean that lambdas are always better than function objects. Function objects are more convenient when their type is required, such as for a declaration of a hash function, sorting, or equivalence criterion of associative or unordered containers. The fact that a function object is usually globally introduced helps to provide them in header files or libraries, whereas lambdas are better for specific behavior specified locally.

10.1.4. Predicates versus Function Objects

Predicates are functions or function objects that return a Boolean value (a value that is convertible into `bool`). However, not every function that returns a Boolean value is a valid predicate for the STL. This may lead to surprising behavior. Consider the following example:

Click here to view code image

```
// fo/removeif1.cpp

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

class Nth {     // function object that returns true for the nth call
  private:
    int nth;        // call for which to return true
    int count;      // call counter
  public:
    Nth (int n) : nth(n), count(0) {
    }
    bool operator() (int) {
        return ++count == nth;
    }
};

int main()
{
    list<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    PRINT_ELEMENTS(coll,"coll:         ");

    // remove third element
    list<int>::iterator pos;
    pos = remove_if(coll.begin(),coll.end(),  // range
                    Nth(3));                  // remove criterion
    coll.erase(pos,coll.end());

    PRINT_ELEMENTS(coll,"3rd removed: ");
}
```

This program defines a function object `Nth` that yields `true` for the *n*th call. However, when passing it to `remove_if()`, an algorithm that removes all elements for which a unary predicate yields `true` (see Section 11.7.1, page 575), the result is a big surprise:[3]

[3] At least this is the output of gcc 4.5 and Visual C++ 2010, but other platforms might result in a different output; see the following discussion.

```
coll:         1 2 3 4 5 6 7 8 9 10
3rd removed: 1 2 4 5 7 8 9 10
```

Two elements, the third and sixth elements, are removed. This happens because the usual implementation of the algorithm copies the predicate internally during the algorithm:

```
template <typename ForwIter, typename Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end,
                        Predicate op)
{
    beg = find_if(beg, end, op);

    if (beg == end) {
        return beg;
    }
    else {
        ForwIter next = beg;
        return remove_copy_if(++next, end, beg, op);
    }
}
```

The algorithm uses `find_if()` to find the first element that should be removed. However, the algorithm then uses a copy of the passed predicate `op` to process the remaining elements, if any. Here, `Nth` in its original state is used again and also removes the third element of the remaining elements, which is in fact the sixth element.

This behavior is not a bug. The standard does not specify how often a predicate might be copied internally by an algorithm. Thus, to get the guaranteed behavior of the C++ standard library, you should not pass a function object for which the behavior depends on how often it is copied or called. Thus, if you call a unary predicate for two arguments and both arguments are equal, the predicate should always yield the same result.

In other words: **A predicate should always be stateless**. That is, a predicate should not change its state due to a call, and a copy of a predicate should have the same state as the original. To ensure that you can't change the state of a predicate due to a function call, you should declare operator `()` as a constant member function.

It is possible to avoid this surprising behavior and to guarantee that this algorithm works as expected even for a function object such as `Nth`, without any performance penalties. You could implement `remove_if()` in such a way that the call of `find_if()` is replaced by its contents:

```
template <typename ForwIter, typename Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end,
                        Predicate op)
{
    while (beg != end && !op(*beg)) {
        ++beg;
    }
    if (beg == end) {
        return beg;
    }
    else {
        ForwIter next = beg;
       return remove_copy_if(++next, end, beg,  op);
    }
}
```

So, it might be a good idea to change the implementation of `remove_if()` or submit a change request to the implementer of the library. To my knowledge, this problem arises in current implementations only with the `remove_if()` algorithm. If you use `remove_copy_if()`, all works as expected. (Whether the C++ standard library should guarantee the expected behavior in cases such as those presented in this example was under discussion but never changed.) However, for portability, you should never rely on this implementation detail. You should always declare the function call operator of predicates as being a constant member function.

Note that with lambdas, you can share the state among all copies of the function object, so this problem doesn't apply. See Section 10.3.2, page 501, for details.