8.1 The Concept of Function Objects

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

```cpp
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,

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

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

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

This kind of definition is more complicated; however, it has three important advantages:

1. A function object might be smarter because it may have a state. In fact, you can have two instances of the same function, represented by a function object, 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 page 126 for more details about these advantages and page 127 for 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 another subsection.

8.1.1 Function Objects as Sorting Criteria

Programmers often need a sorted collection of elements that have a special class (for example, a collection of persons). However, you either don't want to use or you 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:

```cpp
// 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
   * - if the last name is
   * equal and the first name is less
   */
return p1.lastname() < p2.lastname() ||
   (! (p2.lastname() < p1.lastname()) &&
    p1.firstname() < p2.firstname());
};

int main()
{
    typedef set<Person, PersonSortCriterion> PersonSet;
    PersonSet coll;
    ...

    //do something with the elements
    PersonSet::iterator pos;
    for (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 Persons according to their last name and (if they are equal) to
their first name. 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 on page 123).

All sets with this sorting criterion have their own type (which is called PersonSet in this
example). 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 the next subsection). See page 178 for more details about sets and their sorting criteria.

8.1.2 **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:

```cpp
// fo/general.cpp

#include <iostream>
#include <list>
#include <algorithm>
#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;

   // insert values from 1 to 9
   generate_n (back_inserter(coll),
      // start
In this example, a function object is used that 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. These algorithms 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 `++coll.begin()` up to `--coll.end()`.[1] The output of the program is as follows:

[1] The expressions

```
++coll.begin()
```

and

```
--
coll.end()
```

might not work with vectors. This nasty problem is discussed in Section 7.2.6.

```
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 produce more complicated sequences easily.

**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 two ways to get a "result" or "feedback" from using function objects with algorithms:

1. You can pass the function objects by reference.
2. You can use the return value of the `for_each()` algorithm.

The latter 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. For example:

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

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

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

    //"function call"
    int operator() () { 
```
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),
        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:

```cpp
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:

```
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.
8.1.3 The Return Value of `for_each()`

The effort involved with a reference-counted implementation of a function object 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:

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

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);  
    }
};
```

//fo/foreach3.cpp
int main()
{
    vector<int> coll;

    //insert elements from 1 to 8
    for (int i=1; i<=8; ++i) {
        coll.push_back(i);
    }

    //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 it 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. Then you could use the mean value that is processed by for_each() directly. See page 336 for such an example.
8.1.4 Predicates versus Function Objects

Predicates are functions or function objects that return a Boolean value (a value that is convertible to `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:

```cpp
// fo/removeif.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;

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

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

```
This program defines a function object \( \text{Nth} \) that yields \( \text{true} \) for the \( n \)th call. However, when passing it to \text{remove_if()} \) (an algorithm that removes all elements for which a unary predicate yields \( \text{true} \), see page 378), the result is a big surprise:

\[
\text{coll: } 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \\
\text{nth removed: } 1 \ 2 \ 4 \ 5 \ 7 \ 8 \ 9
\]

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

```
template <class ForwIter, class Predicate>
ForwIter std::remove_if(ForwIter beg, ForwIter end, 
    Predicate \( \text{op} \))
{
    beg = find_if(beg, end, \( \text{op} \));
    if (beg == end) 
    
    }
    else {
        ForwIter next = beg;
        return remove_copy_if(++next, end, beg, \( \text{op} \));
    }
}
```

The algorithm uses \( \text{find_if()} \) to find the first element that should be removed. However, it then uses a copy of the passed predicate \( \text{op} \) to process the remaining elements if any. Here, \( \text{Nth} \) in its original state is used again and it 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, then the predicate should always yield the same result. 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 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:

```cpp
template <class ForwIter, class 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 implementor of the library). To my knowledge, in current implementations this problem only arises with the `remove_if()` algorithm. If you use `remove_copy_if()`, all works as expected. However, to be portable, you should never rely on this implementation detail. You should always declare the function call operator of predicates as being a constant member function.

[3] Whether the C++ standard library should guarantee the expected behavior in cases such as
those presented in this example is currently under discussion.

**URL** [http://search.safaribooksonline.com/0201379260/ch08lev1sec1](http://search.safaribooksonline.com/0201379260/ch08lev1sec1)