“Advanced C++:
Templates and Generic Programming”

Function Objects

Hüseyin Akcan
huseyin.akcan@ieu.edu.tr

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().

- Function objects have 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.
Function objects as sorting criteria

```cpp
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() ||
               (! (p2.lastname() < p1.lastname()) &&
                p1.firstname() < p2.firstname());
    }
};

typedef set<Person,PersonSortCriterion> PersonSet;
```
### Predefined function objects

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>negate&lt;type&gt;()</td>
<td>- param</td>
</tr>
<tr>
<td>plus&lt;type&gt;()</td>
<td>param1 + param2</td>
</tr>
<tr>
<td>minus&lt;type&gt;()</td>
<td>param1 - param2</td>
</tr>
<tr>
<td>multiplies&lt;type&gt;()[^1]</td>
<td>param1 * param2</td>
</tr>
<tr>
<td>divides&lt;type&gt;()</td>
<td>param1 / param2</td>
</tr>
<tr>
<td>modulus&lt;type&gt;()</td>
<td>param1 % param2</td>
</tr>
<tr>
<td>equal_to&lt;type&gt;()</td>
<td>param1 == param2</td>
</tr>
<tr>
<td>not_equal_to&lt;type&gt;()</td>
<td>param1 != param2</td>
</tr>
<tr>
<td>less&lt;type&gt;()</td>
<td>param1 &lt; param2</td>
</tr>
<tr>
<td>greater&lt;type&gt;()</td>
<td>param1 &gt; param2</td>
</tr>
<tr>
<td>less_equal&lt;type&gt;()</td>
<td>param1 &lt;= param2</td>
</tr>
<tr>
<td>greater_equal&lt;type&gt;()</td>
<td>param1 &gt;= param2</td>
</tr>
<tr>
<td>logical_not&lt;type&gt;()</td>
<td>! param</td>
</tr>
<tr>
<td>logical_and&lt;type&gt;()</td>
<td>param1 &amp;&amp; param2</td>
</tr>
<tr>
<td>logical_or&lt;type&gt;()</td>
<td>param1</td>
</tr>
</tbody>
</table>

[^1]: multiplies<type>() is available in C++11 and later.
Function adapters

- A function adapter is a function object that enables the combining of function objects with each other, with certain values, or with special functions. For example, in the following statement:

```cpp
find_if (coll.begin(), coll.end(),        // range
        bind2nd (greater<int>(), 42))    // criterion
```

produces a combined function object that checks whether an int value is greater than 42. In fact, bind2nd transforms a binary function object, such as `greater<>`, into a unary function object. It always uses its second parameter as the second argument of the binary function object that is passed as the first parameter.

### Table: Function Adapters

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bind1st(op, value)</code></td>
<td><code>op(value, param)</code></td>
</tr>
<tr>
<td><code>bind2nd(op, value)</code></td>
<td><code>op(param, value)</code></td>
</tr>
<tr>
<td><code>not1(op)</code></td>
<td><code>!op(param)</code></td>
</tr>
<tr>
<td><code>not2(op)</code></td>
<td><code>!op(param1, param2)</code></td>
</tr>
</tbody>
</table>

```cpp
pos = find_if (coll.begin() , coll.end(), not1(bind2nd(modulus<int>(), 2)));
// returns the first even element of a collection
```
Function adapters for member functions

- The C++ standard library provides some additional function adapters that enable you to call a member function for each element of a collection.

```cpp
class Person {
    private:
        std::string name;
    public:
        void print() const {
            std::cout << name << std::endl;
        }
        void printWithPrefix (std::string prefix) const {
            std::cout << prefix << name << std::endl;
        }
};

void foo (const std::vector<Person>& coll) {
    using std::for_each;
    using std::bind2nd;
    using std::mem_fun_ref;

    //call member function print() for each element
    for_each (coll.begin(), coll.end(),
        mem_fun_ref(&Person::print));

    //call member function printWithPrefix() for each element
    //"person: " is passed as an argument to the member function
    for_each (coll.begin(), coll.end(),
        bind2nd (mem_fun_ref (&Person::printWithPrefix),
            "person: "));
}
Function adapters for ordinary functions

- `ptr_fun` function adapter enables ordinary functions to be used from other function adapters

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ptr_fun(op)</code></td>
<td><code>*op(param)</code></td>
</tr>
<tr>
<td></td>
<td><code>*op(param1, param2)</code></td>
</tr>
</tbody>
</table>

bool check(int elem);

pos = find_if (coll.begin(), coll.end(), not1(ptr_fun(check)));

The second form is used when you have a global function for two parameters and, for example, you want to use it as a unary function:

```c
//find first string that is not empty
pos = find_if (coll.begin(), coll.end(), bind2nd(ptr_fun(strcmp),""));
```
User defined function objects for function adapters

You can write your own function objects, but to use them in combination with function adapters they must meet certain requirements: They must provide type members for the type of their arguments and the result. The C++ standard library provides structures to make this more convenient:

```cpp
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```
User defined function objects for function adapters

Thus, by deriving your function object from one of these types you meet the requirements easily so that your function object becomes "adapter-able."

The following example shows a complete definition for a function object that processes the first argument raised to the power of the second argument:

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

//print 3 raised to the power of all elements
transform (coll.begin(), coll.end(),
           ostream_iterator<int>(cout," "),
           bind1st(fopow<float, int>(),3));
```

3 9 27 81 243 729 2187 6561 19683
Limitations of STL function objects

- The ability to compose function objects is important for building software components from other components.

- It enables you to construct very complicated function objects from simple ones.

- So in general it should be possible to define almost every functional behavior as a combination of function objects.

- However, the C++ standard library does not provide enough adapters to support this.

- For example, it is not possible to combine the result of two unary operations to formulate a criterion such as "this and that."