
10.3. Using Lambdas

As introduced in Section 3.1.10, page 28, lambdas were introduced with C++11. They provide a powerful and very convenient way to provide local functionality, especially to specify details of algorithms and member functions. Although lambdas are a language feature, their use is so important for the C++ standard library that I will go into a couple of details here.

As introduced in Section 6.9, page 229, lambdas provide significant improvements for C++ when using the STL because now you have an intuitive, readable way to pass individual behavior to algorithms and container member functions. If you need specific behavior passed to an algorithm, just specify it like any other function right there where you need it.

The best way to demonstrate the use of lambdas is by example, especially when comparing corresponding code not using lambdas. In the following subsections, I provide some examples of functionality introduced before with other function objects and adapters, such as `bind()`.

10.3.1. Lambdas versus Binders

Take, for example, *fo/bind1.cpp*, which is presented in Section 10.2.2, page 488. When lambdas are used, the corresponding code looks as follows:

Click here to view code image

```
// fo/lambda1.cpp

#include <iostream>

int main()
{
    auto plus10 = [] (int i) {
                      return i+10;
                  };
    std::cout << "+10:     " << plus10(7) << std::endl;

    auto plus10times2 = [] (int i) {
                          return (i+10)*2;
                        };
    std::cout << "+10 *2: " << plus10times2(7) << std::endl;

    auto pow3 = [] (int i) {
                    return i*i*i;
                };
    std::cout << "x*x*x:   " << pow3(7) << std::endl;

    auto inversDivide = [] (double d1, double d2) {
                            return d2/d1;
                        };
    std::cout << "invdiv: " << inversDivide(49,7) << std::endl;
}
```

Just to compare one function object declaration: Declaring to "add `10` and multiply by `2`" looks with binders as follows:

Click here to view code image

```
auto plus10times2 = std::bind(std::multiplies<int>(),
                              std::bind(std::plus<int>(),
                                        std::placeholders::_1,
                                        10),
                              2);
```

The same functionality defined with lambdas looks as follows:

```
auto plus10times2 = [] (int i) {
                        return (i+10)*2;
                    };
```

10.3.2. Lambdas versus Stateful Function Objects

Let's now replace a custom function object by a lambda. Consider the example to process the mean value of elements in Section 10.1.3, page 482. A version with lambdas looks as follows:

Click here to view code image

```
// fo/lambda2.cpp

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

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

   // process and print mean value
   long sum = 0;
   for_each (coll.begin(), coll.end(),     // range
             [&sum] (int elem) {
                 sum += elem;
             });
   double mv = static_cast<double>(sum)/static_cast<double>
(coll.size());
   cout << "mean value: " << mv << endl;
}
```

Here, instead of the need to define a class for the function object passed, you simply pass the required functionality. However, the state of the calculation is held outside the lambda in `sum`, so you ultimately have to use `sum` to compute the mean value.

With a function object, this state (`sum`) is entirely encapsulated, and we can provide additional member functions to deal with the state (such as to process the mean value out of `sum`).

Click here to view code image

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

So, from a calling point of view, you can consider the user-defined function object as being more condensed and less error-prone than the lambda version presented here.

When dealing with state, you should also be careful when using `mutable`. Consider the example introduced in Section 10.1.4, page 483, in which your search criterion is a stateful function object searching for the third element. A corresponding version using lambdas should, strictly speaking, pass the internal counter, which represents its state, by value because the counter is not needed outside the algorithm called. By using `mutable`, you could provide write access to this state then for all "function calls":

Click here to view code image

```
// fo/lambda3.cpp

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

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;
    int count=0;      // call counter
    pos = remove_if(coll.begin(),coll.end(),     // range
                    [count] (int) mutable {      // remove criterion
                       return ++count == 3;
                    });
    coll.erase(pos,coll.end());

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

However, as described in Section 10.1.4, page 483, you can then run into the problem that two elements, the third and the sixth, get removed, which results in the following output:

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

Again, the reason for this output is that the lambda object gets copied by the `remove_if()` algorithm while it is running, so two lambda objects exist that both remove the third element. Thus, the state gets duplicated.

If you pass the argument by reference and don't use `mutable`, the behavior is as expected, because both lambda objects internally used by `remove_if()` share the same state. Thus, with the following:

```
int count=0;      // call counter
pos = remove_if(coll.begin(),coll.end(),   // range
                [&count] (int) {           // remove criterion
                    return ++count == 3;
                });
```

the output is:

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

10.3.3. Lambdas Calling Global and Member Functions

Of course, any lambda can call other functions, so the lambda version of *fo/compose3.cpp* in Section 10.2.2, page 490, looks as follows:

Click here to view code image

```
// fo/lambda4.cpp

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

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
                  [] (char c1, char c2) {      // compar. criterion
                      return myToupper(c1)==myToupper(c2);
                  });
    if (pos != s.end()) {
        cout << "\"" << sub << "\" is part of \"" << s << "\""
             << endl;
    }
}
```

Of course, you can call member functions the same way (compare with Section 10.2.2, page 491):

Click here to view code image

```
// fo/lambda5.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(),
              [] (const Person& p) {
                  p.print();
              });
    cout << endl;

    // call member function print2() with additional argument for each person
    for_each (coll.begin(), coll.end(),
              [] (const Person& p) {
                  p.print2("Person: ");
              });
}
```

10.3.4. Lambdas as Hash Function, Sorting, or Equivalence Criterion

As mentioned before, you can also use lambdas as hash functions, ordering, or sorting criteria. For example:

```
class Person {
    ...
};

auto hash = [] (const Person& p) {
    ...
};
auto eq = [] (const Person& p1, const Person& p2) {
    ...
};

// create unordered set with user-defined behavior
unordered_set<Person,decltype(hash),decltype(eq)> pset(10,hash,eq);
```

Note again that you have to use `decltype` to pass the type of the lambda to the `unordered_set` because it creates its own instance of them. In addition, you have to pass a hash function and equivalence criterion to the constructor because otherwise, the constructor calls the default constructor for the hash function and equivalence criterion, which is not defined for lambdas.

Due to these inconveniences, specifying a class for the function objects here can be considered as being more readable and even more convenient. So when state is involved, lambdas are not always better.

See Section 7.9.7, page 379, for a complete example of how to use lambdas to specify a hash function and an equivalence criterion for unordered containers.