
6.9. Using Lambdas

Lambdas, introduced with C++11, define a way to specify functional behavior inside an expression or statement (see Section 3.1.10, page 28). As a consequence, you can define objects that represent functional behavior and pass these objects as inline argument to algorithms to be used as predicates or for other purposes.

For example, in the following statement:

Click here to view code image

```
// transform all elements to the power of 3
std::transform (coll.begin(), coll.end(),      // source
                coll.begin(),                  // destination
                [](double d) {                 // lambda as function object
                    return d*d*d;
                });
```

the expression

```
[](double d) { return d*d*d; }
```

defines a lambda expression, which represents a function object that returns a `double` raised to the power of 3. As you can see, this provides the ability to specify the functional behavior passed to `transform()` directly where it is called.

The Benefit of Lambdas

Using lambdas to specify behavior inside the STL framework solves a lot of drawbacks of previous attempts. Suppose that you search in a collection for the first element with a value that is between `x` and `y`:

Click here to view code image

```
// stl/lambda1.cpp

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

int main()
{
    deque<int> coll = { 1, 3, 19, 5, 13, 7, 11, 2, 17 };
    int x = 5;
    int y = 12;
    auto pos = find_if (coll.cbegin(), coll.cend(),    // range
                        [=](int i) {                   // search criterion
                            return i > x && i < y;
                        });
    cout << "first elem >5 and <12: " << *pos << endl;
}
```

When calling `find_if()`, you pass the corresponding predicate inline as the third argument:

```
auto pos = find_if (coll.cbegin(), coll.cend(),
                    [=](int i) {
                        return i > x && i < y;
                    });
```

The lambda is just a function object taking an integer `i` and returning whether it is greater than `x` and less than `y`:

```
[=](int i) {
    return i > x && i < y;
}
```

By specifying `=` as a *capture* inside `[=]`, you pass the symbols, which are valid where the lambda gets declared, *by value* into the body of the lambda. Thus, inside the lambda, you have read access to the variables `x` and `y` declared in `main()`. With `[&]`, you could even pass the values by reference so that inside the lambda, you could modify their values (see Section 3.1.10, page 29, for more details about captures).

Now compare this way to search for "the first element `>5` and `<12`" with the other approaches provided by C++ before lambdas

were introduced:

- In contrast to using handwritten loops:

Click here to view code image

```
// find first element > x and < y
vector<int>::iterator pos;
for (pos = coll.begin() ; pos != coll.end(); ++pos) {
    if (*pos > x && *pos < y) {
        break; // the loop
    }
}
```

you benefit from using predefined algorithms and avoid a more or less ugly `break` .

- In contrast to using a self-written function predicate:

Click here to view code image

```
bool pred (int i)
{
    return i > x && i < y;
}
...
pos = find_if (coll.begin(), coll.end(),      // range
               pred);                         // search criterion
```

you don't have the problem that the details of the behavior are written somewhere else and you have to scroll up to find out what `find_if()` exactly is looking for, unless you have and trust a corresponding comment. In addition, C++ compilers optimize lambdas better than they do ordinary functions.

What's more, access to `x` and `y` becomes really ugly in this scenario. The usual solution before C++11 to use a function object (see Section 6.10, page 233) demonstrates the whole ugliness of this approach:

Click here to view code image

```
class Pred
{
  private:
    int x;
    int y;
  public:
    Pred (int xx, int yy) : x(xx), y(yy) {
    }
    bool operator() (int i) const {
        return i > x && i < y;
    }
};
...
pos = find_if (coll.begin(), coll.end(),      // range
               Pred(x,y));                    // search criterion
```

- In contrast to using binders (introduced in Section 6.10.3, page 241):

Click here to view code image

```
pos = find_if (coll.begin(), coll.end(),          // range
               bind(logical_and<bool>(),          // search criterion
                    bind(greater<int>(),_1,x),
                    bind(less<int>(),_1,y)));
```

you don't have problems understanding the expression defined here.

To summarize, lambdas provide the first convenient, readable, fast, and maintainable approach to use STL algorithms.

Using Lambdas as Sorting Criterion

As another example, let's use a lambda expression to define the criterion when sorting a vector of `Person` s (see Section 6.8.2, page 228, for a corresponding program that uses a function to define the sorting criterion):

Click here to view code image

```
// stl/sort2.cpp

#include <algorithm>
#include <deque>
#include <string>
#include <iostream>
using namespace std;
```

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

int main()
{
    deque<Person> coll;
    ...

    // sort Persons according to lastname (and firstname):
    sort(coll.begin(),coll.end(),                    // range
         [] (const Person& p1, const Person& p2) { // sort criterion
              return p1.lastname()<p2.lastname() ||
                     (p1.lastname()==p2.lastname() &&
                      p1.firstname()<p2.firstname());
         });
    ...
}
```

Limits of Lambdas

However, lambdas are not better in every case. Consider, for example, using a lambda to specify the sorting criterion for associative containers:

Click here to view code image

```
auto cmp = [] (const Person&   p1, const Person&  p2) {
                return p1.lastname()<p2.lastname() ||
                       (p1.lastname()==p2.lastname()&&
                        p1.firstname()<p2.firstname());
           };
...
std::set<Person,decltype(cmp)> coll(cmp);
```

Because you need the type of the lambda for the declaration of the `set`, `decltype` (see Section 3.1.11, page 32) must be used, which yields the type of a lambda object, such as `cmp`. Note that you also have to pass the lambda object to the constructor of `coll`; otherwise, `coll` would call the default constructor for the sorting criterion passed, and by rule lambdas have no default constructor and no assignment operator.[12] So, for a sorting criterion, a class defining the function objects might still be more intuitive.

[12] Thanks to Alisdair Meredith for pointing this out.

Another problem of lambdas is that they can't have an internal state held over multiple calls of a lambda. If you need such a state, you have to declare an object or variable in the outer scope and pass it by-reference with a capture into the lambda. In contrast, function objects allow you to encapsulate an internal state (see Section 10.3.2, page 500, for more details and examples).

Nevertheless, you can also use lambdas to specify a hash function and/or equivalence criterion of unordered containers. See Section 7.9.7, page 379, for an example.