11.4. The `for_each()` Algorithm

The `for_each()` algorithm is very flexible because it allows you to access, process, and modify each element in many different ways. Note, however, that since C++11, the range-based `for` loop provides this behavior more conveniently and more naturally (see Section 3.1.4, page 17, and Section 6.2.1, page 174). Thus, `for_each()` might lose its importance over time.

`UnaryProc`

for _ each (InputIterator *beg*, InputIterator *end*, UnaryProc *op*)

- Calls

  op(*elem*)

  for each element in the range [*beg* , *end*) .
- Returns the (internally modified) copy of *op*. Since C++11, the returned *op* is moved.
- *op* might modify the elements. However, see Section 11.2.2, page 509, for a comparison with the `transform()` algorithm, which is able to do the same thing in a slightly different way.
- Any return value of *op* is ignored.
- See Section 6.10.1, page 235, for the implementation of the `for_each()` algorithm.
- Complexity: linear (*numElems* calls of *op* ()).

The following example of `for_each()` passes each element to a lambda that prints the passed element. Thus, the call prints each element:

Click here to view code image

```
// algo/foreach1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);

    // call print() for each element
    for_each (coll.cbegin(), coll.cend(),       // range
              [](int elem){                     // operation
                  cout << elem << ' ';
              });
    cout << endl;
}
```

The program has the following output:

```
1 2 3 4 5 6 7 8 9
```

Instead of a lambda, you could also pass an ordinary function, which is called for each element:

Click here to view code image

```
void print (int elem)
{
    cout << elem << ' ';
}
...
for_each (coll.cbegin(), coll.cend(),  // range
          print);                      // operation
```

But note again that since C++11, using a range-based `for` loop is often more convenient:

```
for (auto elem : coll) {
    cout << elem << ' ';
}
```

The following example demonstrates how to modify each element:

Click here to view code image

```
// algo/foreach2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);

    // add 10 to each element
    for_each (coll.begin(), coll.end(),          // range
              [](int& elem){                     // operation
                  elem += 10;
              });
    PRINT_ELEMENTS(coll);

    // add value of first element to each element
    for_each (coll.begin(), coll.end(),          // range
              [=](int& elem){                    // operation
                  elem += *coll.begin();
              });
    PRINT_ELEMENTS(coll);
}
```

The program has the following output:

```
11 12 13 14 15 16 17 18 19
22 23 24 25 26 27 28 29 30
```

As you can see, you have to declare the `elem` to be a reference in order to modify it and to define a capture, such as `[=]` , to be able to add a copy of the first element:

Click here to view code image

```
for_each (coll.begin(), coll.end(),          // range
          [=](int& elem){                    // operation
              elem += *coll.begin();
          });
```

If instead you passed a reference to the first element with the second call of `for_each()` :

Click here to view code image

```
for_each (coll.begin(), coll.end(),          // range
          [&](int& elem){                    // operation
              elem += *coll.begin();
          });
```

the value to add would change while the elements are processed, which would result in the following output:

```
11 12 13 14 15 16 17 18 19
22 34 35 36 37 38 39 40 41
```

You could also define an ordinary function object:

```
// function object that adds the value with which it is initialized
template <typename T>
class AddValue {
  private:
    T theValue;      // value to add
  public:
    // constructor initializes the value to add
    AddValue (const T& v) : theValue(v) {
    }

    // the function call for the element adds the value
    void operator() (T& elem) const {
        elem += theValue;
    }
};
```

and pass it to `for_each()` :

Click here to view code image

```
for_each (coll.begin(), coll.end(),          // range
          AddValue<int>(10));                // operation
...
for_each (coll.begin(), coll.end(),          // range
          AddValue<int>(*coll.begin()));     // operation
```

The AddValue<> class defines function objects that add a value to each element that is passed to the constructor. See Section 6.10.1, page 237, for more details about this example.

Note also that you can do the same by using the transform() algorithm (see Section 11.6.3, page 563) in the following way:

Click here to view code image

```
// add 10 to each element
transform (coll.cbegin(), coll.cend(),          // source range
           coll.begin(),                         // destination range
           [](int elem){                        // operation
               return elem + 10;
           });
...

// add value of first element to each element
transform (coll.cbegin(), coll.cend(),          // source range
           coll.begin(),                         // destination range
           [=](int elem){                        // operation
               return elem + *coll.begin();
           });
```

See Section 11.2.2, page 509, for a general comparison between for_each() and transform() .

A third example demonstrates how to use the return value of the for_each() algorithm. Because for_each() has the special property that it returns its operation, you can process and return a result inside the operation:

Click here to view code image

```
// algo/foreach3.cpp

#include "algostuff.hpp"
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 (implicit type conversion)
    operator double() {
        return static_cast<double>(sum) / static_cast<double>(num);
    }
};

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,8);

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

The program has the following output:

```
mean value: 4.5
```

You could also use a lambda and pass the value to return by reference. However, in this scenario, a lambda is not necessarily better, because a function object really encapsulates both `sum` as internal state and the final division of dividing the sum by the number of elements. See Section 10.1.3, page 482, for details.