IB9N7 C++ for Quantitative Finance

Lecture 15: STL part 2

J.F.Reizenstein@warwick.ac.uk

25 February 2016
(Week 21)

Outline

In this lecture, we'll look at more aspects of using the Standard Template Library:

- lambda functions;
- associative containers;

Lambdas

Lambdas

A Lambda is effectively a way to define a function within another function. In fact, in the following program, `a1` and `a2` are effectively the same thing.

```
1  int main(){
2      auto a1 = [](int x) -> double {
3          int j = x + 2;
4          return j+9;
5      };
6      class A{//allowed but strange to define a class inside a function
7      public:
8          double operator()(int x) const {
9              int j = x+2;
10             return j+9;
11         }
12     };
13     A a2;
14     std::cout<<a1(3)<<"␣"<<a2(3)<<"\n";
15 }
```

notes on lambdas

If the lambda contains only one statement which is a `return` then the compiler can work out the return type for itself. The following are equivalent.

```
auto m = [](int x, int y){return x+y;};
auto n = [](int x, int y) -> int {return x+y;};
```

If the lambda returns void, then the return type can also be omitted.

```
auto a1 = [](int x){
    if (x==4)
        return;
    std::cout<<"I didn't get 4\n";
};
```

If the return type is omitted, and no parameters are needed, then the parameter list can also be omitted.

```
auto a3 = []{return 7;};
```

Note the presence of `;` characters both inside and outside the lambda.

Lambdas with STL

Lambdas provide an easy way to supply functions to the STL. For example, the following sorts a vector of tuples on just the second part.

```
void f(std::vector<std::tuple<int, int>>& v){
    using T = std::tuple<int,int>;
    std::sort(v.begin(),v.end(),[](const T& a, const T& b){
        return std::get<1>(a) < std::get<1>(b);
    }); //note the brackets here
}

```

Two ways to iterate over a vector:

```
void print(const std::vector<int>& v){
    std::for_each(v.begin(),v.begin(),[](int i){ //or (const int& i)
        std::cout<<i<<"\n";
    });
    for(int i : v){ //or const int& i
        std::cout<<i<<"\n";
    }
}
```

The second is simpler, the first is more flexible.

any

The following both check if anyone has a surname longer than 10 characters.

```
class Name{
public:
    std::string m_firstname, m_surname;
};

bool anyLongNames(const std::vector<Name>& v){
    return std::any_of(v.begin(),v.end(),[](const Name& n){
        return n.m_surname.size()>10;
    });
}
bool anyLongNames1(const std::vector<Name>& v){
    for(auto& n : v){
        if (n.m_surname.size()>10){
            return true;
        }
    }
    return false;
}
```

The second may be simpler, but the first may be cleaner in the context of a longer function.

Lambda capture

(Advanced material) The lambda object can be given member variables which are copies of variables in the enclosing function, by naming them inside the brackets. The name can be preceded by `&` to indicate a reference is taken.

```
int main(){
    int x = 13, y=18;
    auto f = [x](int i){return x+i;};
    x = 97;
    std::cout<<f(1)<<"\n";
}
```

Lambda capture 2

The following are equivalent

```
void a(){
    int x = 13, y=18;
    auto f = [x,&y](int i){return x+y+i;};
    x = y = 97;
    std::cout<<f(1)<<"\n";
}
void b(){
    int x = 13, y=18;
    class A{
        int x;
        int &y;
    public:
        A(int x_, int& y_):x(x_), y(y_){}
        int operator()(int i) const {return x+y+i;}
    };
    A f(x,y);
    x = y = 97;
    std::cout<<f(1)<<"\n";
}
```


Lambda capture 3

- To allow the lambda object to capture anything by value, you can just put `=` in the brackets. To allow anything by reference, you just put `&` in the brackets.
- To allow the lambda function to modify its member variables the keyword `mutable` is added. This makes the overloaded operator not be const.

```
void a(){
    int x = 13;
    auto f = [x]() mutable -> int {return ++x;};
}
```

- If a lambda is going to escape its enclosing scope, it will usually not capture by reference.

```
std::function<int(int)> makeMultiplier(int scalar){
    return [=](int a){return scalar * a;}; //[&] would be very bad here
}
```

- When defining a lambda inside a non-static member function, you can specify `this` in the capture list to make the lambda know about the whole of its object.

STL sets and maps

sets

The STL set is a simple associative container. It uses a strict weak ordering to store its elements in ascending order. It will never contain two equivalent elements. For example.

```
#include<set>
#include<iostream>
int main(){
    std::set<int> s;
    s.insert(45);
    s.insert(-1);
    s.insert(5);
    s.insert(45);//This has no effect
    for(int i : s)
        std::cout<<i<<"\n";
}
```

sets 2

- Elements of a set cannot be modified - otherwise the order could be made wrong.
- `set` is easier to use than a sorted vector but often slower due to its memory layout.
- To check if a value `x` is in the set, use `s.count(x)`. To get an iterator to it, use `find`, which will return `end` on failure.
- Set has a `lower_bound` member function just like sorted lists.
- The simple `insert` function used above returns a pair containing an iterator to the element along with a `bool` indicating whether it was already there.

STL Maps

The STL map is perhaps the most important associative container in C++.

Maps

A map is so called because it maps unique keys to values.

A map has the following properties:

- Keys are unique, values need not be.
- Optimised for the random access of elements *by their key*.
- Requires the header `<map>`.

Creating maps

A `std::map` has two template parameters:

1. the first for the **key**;
2. the second for the **value**.

Declaring maps

```
std::map<KeyT, ValueT> identifier;
```

(`KeyT` has to support the < operator, defining a total order on the keys.)

A `std::map<K, V>` is basically the same as a `std::set<std::pair<const K,V>>` where the sorting is just on the first element (the key), except that the value part of each element can be modified.

Elements of maps (1)

To access elements in a map, one can use iterators or the subscript (indexing) operator with a key:

Example: Mapping student IDs to names

```
1 std::map<int, std::string> studentNames;
2 studentNames[600001] = "Student, One";
3 studentNames[1400001] = "Student, Two";
```

Elements of maps (2)

When assigning to a key within a map:

- If an element with the supplied key is not already in the map, then a new element is created.
- If it is already there, then the new value replaces the old one.

Maps and iterators (1)

Maps can also be iterated over.

Map iterators

```
1  std::map<int, std::string>::iterator iter =
       studentNames.begin();
2  // can dereference to get a pair
3  std::pair<int, std::string> student = *iter; // makes a
       copy
4  // and then use the normal dot operator
5  int number = student.first;
6  std::string name = student.second;
7  // or act directly on the iterator, using pointer syntax
8  // to indicate that we refer to the object the iterator points to
9  // not to the iterator object itself
10 number = iter->first;
11 name = iter->second;
```


Maps and iterators

Map iterators

One can print the names of all the students stored in the map by iterating over the map in either of the following ways:

```
1  for(auto &s : studentNames)
2      std::cout<<s.first<<": "<<s.second<<"\n";
3  for(auto i = studentNames.begin(); i!=studentNames.
       end();++i)
4      std::cout<<i->first<<": "<<i->second<<"\n";
```


Own Comparator

If you want to use a set or map without relying on the < operator, it is easiest to define a class to be the comparator. For example, if you want all odd numbers first, try this.

```
1  #include<set>
2  #include<cmath>
3  #include<iostream>
4  class LessThanWithOddsFirst{
5  public:
6      bool operator() (int a, int b){ //or const &
7          if(std::abs(a-b) % 2) //if exactly one of a and b is odd
8              return a%2; //then a comes first iff a is odd
9          return a<b;
10     }
11 };
12 int main(){
13     std::set<int,LessThanWithOddsFirst> s;
14     for(int i=0; i<10; ++i)
15         s.insert(i);
16     for(int i : s)
17         std::cout << i << "\n";
18 }
```

multiset and multimap

`std::multiset` is just like `set` except it allows multiple equivalent elements. Its `count` function can return values greater than 1. The order of equivalent elements is unspecified.
`std::multimap` allows multiple equivalent elements. If all you need is multiple values for a single key (not just equivalent keys), consider using a map with vectors as values.

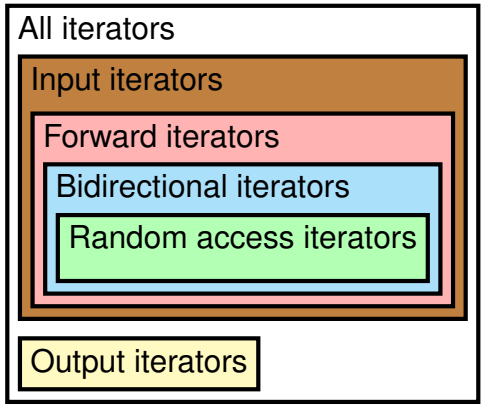
unordered containers

Another variant of all the associative containers is `unordered_set`, `unordered_map`, `unordered_multiset` and `unordered_multimap`. These do not rely on a comparison function, and do not store the values in sorted order. They use a hash function to store elements. They are potentially faster than the sorted versions. The order of elements is unspecified, so you must be careful not to rely on it.

Types of iterators

As we have seen, the `find` algorithm has certain requirements on the iterators it accepts.

There are five important kinds of iterators within the group of all iterators, output iterators being an odd one:



Iterator classifications

Brief description of the different types of iterator:

1. **input iterators**: iterators that support (at least) read dereferencing (as rvalue);
2. **forward iterators**: input iterators for which `iter1 == iter2` implies `++iter1 == ++iter2` (the multi-pass guarantee, i.e. can iterate over the elements more than once with different copies of the iterators);
3. **bidirectional iterators**: forward iterators that also support decrementation. (e.g. `list` iterators)
4. **random access iterators**: bidirectional iterators that also support both incrementation and decrementation by more than one position. (e.g. `vector` iterators)

And finally, the special kind that sits outside the heirarchy:

5. **output iterators**: iterators that support (at least) write dereferencing (as lvalue).

Iterator capability chart ☆

Iterators from the first four kinds that also satisfy the output iterator requirements are called **mutable iterators**.

This chart summarises the necessary and sufficient requirements on iterators in order to fit into each category (with a few technical details omitted):

		All	Input	Forward	Bi-direct-ional	Random access	Output
copy constructible		✓	✓	✓	✓	✓	✓
copy assignable		✓	✓	✓	✓	✓	✓
destructable		✓	✓	✓	✓	✓	✓
dereferenceable †	`*i`	✓	✓	✓	✓	✓	✓
pre-incrementable	`++i`	✓	✓	✓	✓	✓	✓
post-incrementable	`i++`	✓	✓	✓	✓	✓	✓
equality comparable ‡	`==, !=`		✓	✓	✓	✓	
rvalue dereferenceable §	`o = *i, i->member`		✓	✓	✓	✓	
lvalue dereferenceable	`*i = o, *i++ = o`	▽	◊	◊	◊	◊	✓
default constructible				✓	✓	✓	✓
multi-pass possible				✓	✓	✓	
decrementable	`--i, i--`				✓	✓	
supports +, −	`i + n, n + i, i - n, i - j`					✓	
relational comparisons	`i < j, i > j, i <= j, i >= j`					✓	
compound assignment +=, −=	`i += n, i -= n`					✓	
offset dereferencing	`i[n]`					✓	

† if pointing to valid element. see also rvalue vs lvalue
‡ and compatible inequality comparability
§ convertible to value_type
▽ if output iterator or mutable
◊ if mutable

Lab objectives

- Try out STL!