IB9N7 C++ for Quantitative Finance
Worksheet 15

STL2
Hints and solutions

25 February 2016
(Week 21)

Objectives for this lab session

By the end of this session, you should have completed the following:

- Try out STL!

Exercises

Exercise 1: Set using set and unordered_set

Here we write more classes implementing the interface which we has for the objects Set1 and Set2 last week, with two public member functions (i.e. these functions form its interface).

```
void add(int a); //add a to the set
bool isPresent(int a) const; //return whether a is in the set.
```

(a) Create a class Set3 which implements this interface, by storing all the values which have been added in a std::set<int> data member.

(b) Create a class Set4 which implements this interface, by storing all the values which have been added in a std::unordered_set<int> data member.

Here is both done together. The two are identical except for the type of m_data.

```
#include<set>
#include<vector>
#include<unordered_set>
#include<cmath>
#include<iostream>
class Set3{
    std::set<int> m_data;
public:
    void add(int a){
        m_data.insert(a);
    }
    bool isPresent(int a) const{
        return m_data.count(a);
    }
};
class Set4{
    std::unordered_set<int> m_data;
public:
    void add(int a){
        m_data.insert(a);
    }
    bool isPresent(int a) const{
        return m_data.count(a);
    }
};
int main(){
    std::vector<int> a {3,4,5,7,57,12,342};
    Set3 s3; Set4 s4;
    for(auto i : a){
        s3.add(i);
        s4.add(i);
    }
    std::cout<<s3.isPresent(6)<<","<<s3.isPresent(7)<<"\n";
    std::cout<<s4.isPresent(6)<<","<<s4.isPresent(7)<<"\n";
}
```

Exercise 2: Memoization

(a) Implement the following class - by making `calculate` return the factorial of its input. Do not worry about overflow or negative input.

```
class Factorial{
public:
    int calculate(int x); //returns (x!)
};
```

```
class Factorial{
public:
    int calculate(int x); //returns (x!)
};
int Factorial::calculate(int x){
    int ans=1;
    for(int i=1; i<=x; ++i){
        ans*=i;
    }
    return ans;
}
```

(b) Add a private data member `std::map<int,int> m_memo` which the class uses to remember values of the factorial which it has already calculated. Modify the function `calculate` so that it looks for the answer in `m_memo` if available, and only does a calculation if not. If a calculation is done it will be added to `m_memo` as well as returned. This pattern is called memoization and is a common trick.

It makes the code a bit clearer if we move the calculation of the factorial to a separate private function. The following is a straightforward solution.

```
#include <map>
class Factorial{
private:
    std::map<int,int> m_memo;
    static int calculateSimple(int x); //returns (x!)
public:
    int calculate(int x); //returns (x!)
};

int Factorial::calculate(int x){
    if(m_memo.count(x)){
        return m_memo[x];
    }
    int ans = calculateSimple(x);
    m_memo[x] = ans;
    return ans;
}
/*static*/ int Factorial::calculateSimple(int x){
    int ans=1;
    for(int i=1; i<=x; ++i){
        ans*=i;
    }
    return ans;
}
```

There is a slight waste of time because each call to calculate will find the right location in m_memo twice. There are several ways to write a solution which avoids this. Perhaps the neatest, which works in this case because we know that 0 (the default value for new numbers) is never the result of a factorial calculation, is to write it like this.

```
int Factorial::calculate(int x){
    int& ans = m_memo[x];
    if(ans == 0){
        ans = calculateSimple(x);
    }
    return ans;
}
```

I think some of you tried to exploit the fact that if $a > b$ and the value of $b!$ is known, then the calculation of $a!$ can be simplified by starting the loop from b rather than 1. There are some clever things you can do in this direction, but it wasn't the essential point of this exercise. Some approaches might be like these.

```
int Factorial::calculate(int x){
    if(m_memo.count(x)){
        return m_memo[x];
    }
    if(m_memo.empty()){
        m_memo[1]=1;
    }
    if(x<=1){
        return 1;
    }
    auto e = m_memo.end();
    auto& b = *(--e); //the highest known value
    auto ans = b.second;
    for(int i=b.first+1; i<=x; ++i){
        ans*=i;
        m_memo[i]=ans;
    }
    return ans;
}
```

```
int Factorial::calculate(int x){
    if(m_memo.count(x)){
        return m_memo[x];
    }
    if(m_memo.empty()){
        m_memo[1]=1;
    }
    if(x<=1){
        return 1;
    }
    auto e = m_memo.lower_bound(x);
    auto& b = *(--e); //the highest known value less than x
    auto ans = b.second;
    for(int i=b.first+1; i<=x; ++i){
        ans*=i;
    }
    m_memo[x]=ans;
    return ans;
}
```

Exercise 3: Sort with lambda

Write a function which takes a single `std::vector<std::string>` by reference and sorts it so that the elements are in descending order of length. Use a lambda.

Here, with a demonstration.

```
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>
void sortLengthDescending(std::vector<std::string>& v){
    std::sort(v.begin(),v.end(),[](const std::string& a, const std::string& b){
        return a.size() > b.size();
    });
}

int main(){
    std::vector<std::string> n;
    n.push_back("A");
    n.push_back("AAAAAA");
    n.push_back("AB");
    n.push_back("AC");
    n.push_back("E");
    n.push_back("D");
    sortLengthDescending(n);
    for(auto& s : n){
        std::cout<<s<<"\n";
    }
}
```

Exercise 4: Lambda capture

Implement the function which is partially shown here. It takes a vector of people's names and ages and returns the number of people with ages greater than 30.

```
int countOver30s(const std::vector<std::tuple<std::string,int>>& v){
    int total = 0;
    // Fill in here. //
    return total;
}
```

You should iterate over `v` using `std::for_each` with a lambda which captures an appropriate local variable.

```
#include <tuple>
#include <vector>
#include <algorithm>
int countOver30s(const std::vector<std::tuple<std::string,int>>& v){
    int total = 0;
    std::for_each(v.begin(),v.end(),[&total](const std::tuple<std::string,int>& t){
        if(std::get<1>(t) > 30)
            ++total;
    });
    return total;
}

int main(){
    std::vector<std::tuple<std::string,int>> namesAges;
    namesAges.push_back(std::make_tuple("A",43));
    namesAges.push_back(std::make_tuple("B",33));
    namesAges.push_back(std::make_tuple("C",23));
    return countOver30s(namesAges);
}
```

Exercise 5: Other algorithms

Have a look at the other algorithms available at `http://en.cppreference.com/w/cpp/algorithm`.