IB9N7 C++ for Quantitative Finance
Worksheet 16

Templates

3 March 2016
(Week 22)

Exercises

Exercise 1: table by date

In finance, often you know something about certain dates in the future. You might need information for a date which you don't have, in which case there are several things which you might do. Download the example completed project for the Date object and you might want to answer this in `Date_main.cpp`.

- Write a function template according to the following prototype which returns the value in `m` for the date `d`. If there is no such value, return the value for the next date after `d` which is present. If there is no such value, return the value for the latest date in `m`. You can assume `m` is not empty, ideally the function might throw an exception in this case.

```
template<class T>
T& getOnOrAfter(const Date& d, std::map<Date,T>& m);
```

- Write a similar function `getOnOrBefore`. In practice, this is less useful.

Exercise 2: Interpolation using templates

The aim of this exercise is to consider an implementation of interpolation for any function, like in question 2f from worksheet 14. Here is a simplified version, using only the Trapezium rule.

```
#include <iostream>
#include <vector>
#include <cstdlib>

class Integrand
{
    public:
        virtual double operator()(double x) const = 0;
};
class func_square : public Integrand
{
    public:
        double operator()(double x) const
        {
            return x * x;
        }
};

class NIntegrateTrapezium // for numerical integration
{
    public:
        double integrate (const std::vector<double> & grid, const Integrand & f)
const
        {
            double integral = 0.0;
            for (std::vector<double>::size_type i = 1u; i != grid.size(); ++i)
                integral += segment(grid.at(i - 1u), grid.at(i), f);
            return integral;
        }
        double segment (double a, double b, const Integrand & f) const
        {
            return 0.5 * (b - a) * (f(a) + f(b));
        }
};
void integrate_and_print(const NIntegrateTrapezium & quadrature)
{
    std::vector<double> grid(2u);
    grid.at(0u) = 0.0; grid.at(1u) = 1.0;
    func_square f;
    std::cout << quadrature.integrate(grid, f) << std::endl;
}
int main()
{
    NIntegrateTrapezium trapezoidObj;
    integrate_and_print(trapezoidObj);
    return EXIT_SUCCESS;
}
```

- Delete the Integrand class and make the func_square object therefore not inherit from any class.
- Make the member functions `integrate` and `segment` be template functions by adding `template<class Integrand>` just before them.
- Note that the program now works just as it did before, but its structure is very different.

Exercise 3: Template metaprogramming

The following short program illustrates one way to force the compiler to calculate a factorial at compile time. "Programming the compiler" like this is sometimes called metaprogramming. It illustrates class templates, template specialisation, and non-type template parameters. Check you understand how it works.

```
#include<iostream>

template<int i>
class Factorial{
public:
    static const int value = i * Factorial<i-1>::value ;
};

template<>
class Factorial<1>{
public:
    static const int value = 1;
};

int main(){
    std::cout << Factorial<6>::value << "\n";
}
```

Remember to talk through your answers with the lab assistants!