IB9N7 C++ for Quantitative Finance
Worksheet 12

Dynamic polymorphism
Hints and solutions

03 February 2016
(Week 18)

Objectives for this lab session

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

- Work through, modify and extend some examples of virtual inheritance.

Exercises

Exercise 1: 'Simple' examples of virtual inheritance

Consider the following classes and `main` function.
Explain the output it produces.

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

class A
{
    public :
        void f () const {std::cout << "A::f" << std::endl;}
        void g () const {std::cout << "A::g" << std::endl;}
        virtual void h () const {std::cout << "A::h" << std::endl;}
};

class B : public A
{
    public :
        void h () const {std::cout << "B::h" << std::endl;}
    protected :
        void f () const {std::cout << "B::f" << std::endl;}
    private :
        virtual void g () const {std::cout << "B::g" << std::endl;}
};

class C : public B
{
    public :
        void f () const {g();}
        void g () const {h();}
};

void invoke_all(const A & a)
{
    a.f();
    a.g();
    a.h();
}
```

```
35
36  int main ()
37  {
38      ////////////////////////////////////////////////////////////////////
39      // Explain the output of each of the following member function calls
40      ////////////////////////////////////////////////////////////////////
41
42      for (int i = 0; i != 78; ++i) std::cout << "-"; std::cout << std::endl;
43
44      A a;
45      invoke_all(a);
46
47      for (int i = 0; i != 78; ++i) std::cout << "-"; std::cout << std::endl;
48
49      B b;
50      invoke_all(b);
51
52      for (int i = 0; i != 78; ++i) std::cout << "-"; std::cout << std::endl;
53
54      C c;
55      invoke_all(c);
56
57      for (int i = 0; i != 78; ++i) std::cout << "-"; std::cout << std::endl;
58
59      return EXIT_SUCCESS;
60  }
```

Can be determined by running the code: A::f, A::g, A::h, –, A::f, A::g, B::h, –, A::f, A::g, B::h. Be sure that you know why output is what it is.

Exercise 2: Numerical integration

(a) Run the example from the lecture (you may just use one cpp file for everything if you find it easier). You can download the file from the course page.

(b) Try to understand the numerical integrations that the code is performing.

Wikipedia has some good illustrations at `http://en.wikipedia.org/wiki/Numerical_integration`, though of course the grid in our examples is very simple — feel free to improve!

(c) Try it without the keyword `virtual` in the base class `segment` function.

You should notice that Simpson's Rule is never used, and you should be able to explain why (it was mentioned in the lecture).

(d) Implement the midpoint quadrature rule in a new class.

Hint: for the grid interval $[a, b]$, the midpoint rule uses the approximation

$$\int_a^b f \approx (b-a) f\left(\frac{a+b}{2}\right).$$

Again, this is illustrated at `http://en.wikipedia.org/wiki/Numerical_integration`.

```
1  class NIntegrateMidpoint : public NIntegrate
2  {
3      public:
4          double segment (double a, double b) const
5          {
6              return (b - a) * f((a + b) / 2.0);
7          }
8  };
```

(e) Modify the `NIntegrate` class so that it is abstract (by making the `segment` function pure virtual), and provide the trapezium rule functionality instead in a separate derived class.

The `private` access specifier for the `segment` function should be dropped (it was ONLY ever used for illustrating the behaviour of `private` functions in a `virtual` sense and should have been removed as soon as that detail was understood).

```
class NIntegrate // for numerical integration
{
    public:

        double integrate (const std::vector<double> & grid) 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));
            return integral;
        }

        virtual double segment (double a, double b) const = 0;
};

class NIntegrateTrapezium : public NIntegrate
{
    public:
        double segment (double a, double b) const
        {
            return 0.5 * (b - a) * (f(a) + f(b));
        }
};
```

(f) Currently the code only works for the provided function `f`, and so is not sufficiently general.

Define a pure abstract base class `Integrand` to replace the function `f` in the example. You may use any interface you wish (possibly but not necessarily overloading the parenthesis operator).

```
#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 NIntegrate // 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;
        }

        virtual double segment (double a, double b, const Integrand & f) const =
     0;
};

class NIntegrateTrapezium : public NIntegrate
{
    public:
        double segment (double a, double b, const Integrand & f) const
        {
            return 0.5 * (b - a) * (f(a) + f(b));
        }
};

class NIntegrateSimpson : public NIntegrate // Simpson's Rule
{
    public:
        double segment (double a, double b, const Integrand & f) const
        {
            return (b - a) * (f(a) + 4.0 * f(0.5 * (a + b)) + f(b)) / 6.0;
        }
};

class NIntegrateMidpoint : public NIntegrate
{
    public:
        double segment (double a, double b, const Integrand & f) const
        {
            return (b - a) * f((a + b) / 2.0);
        }
};

void integrate_and_print(const NIntegrate & 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;
    NIntegrateSimpson simpsonObj;
    NIntegrateMidpoint midpointObj;

    integrate_and_print(trapezoidObj);
    integrate_and_print(simpsonObj);
    integrate_and_print(midpointObj);

    return EXIT_SUCCESS;
}
```

Exercise 3: Equity Monte Carlo

(a) Download and extract 12_EquityMC.zip, and open the project. This is a simple monte carlo pricer for european equity options. Have a look around and try to understand how it works. Ask for help. (There is some mathematical code here, in the EuropeanOption class the Black Scholes formula is implemented to price European options analytically, and in the Pricer class the Euler step scheme is implemented. I think you have learnt about this in other courses. It is just included to provide a realistic example. Don't focus on it here.)

(b) Implement a basket of options: create a new class `BasketOfOptions` which owns several `EuropeanOption`s which all expire at the same time, and which implements the Payoff interface. Its payoff should be the sum of the payoffs of the options it contains. Do not worry about an analytic price for it. This can be done either with the owned options on the stack or the heap. Here I demonstrate both versions. I add the following to the top of `EquityOption.h`.

```
#include <vector>
#include <memory>
```

the following to the end of that file

```
class BasketOfOptions : public Payoff {
    std::vector<EuropeanOption> m_options;
public:
    void add (const EuropeanOption& o);
    double timeToExpiry() const override;
    double payoffOnExpiry(double equityPrice) const override;
};

class BasketOfOptions2 : public Payoff {
    std::vector<std::unique_ptr<EuropeanOption>> m_options;
public:
    void add (std::unique_ptr<EuropeanOption> o);
    double timeToExpiry() const override;
    double payoffOnExpiry(double equityPrice) const override;
};
```

and the following to `EquityOption.cpp`.

```
void BasketOfOptions::add (const EuropeanOption& o){
    //here we trust the user to add options all with the same expiry.
    m_options.push_back(o);
}

double BasketOfOptions::timeToExpiry() const {
    return m_options.at(0u).timeToExpiry();
}

double BasketOfOptions::payoffOnExpiry(double equityPrice) const {
    double p = 0.0;
    for(const auto& o : m_options){
        p += o.payoffOnExpiry(equityPrice);
    }
    return p;
}

void BasketOfOptions2::add (std::unique_ptr<EuropeanOption> o){
    //here we trust the user to add options all with the same expiry.
    //This swap trick was discussed in a previous worksheet
    m_options.push_back(std::unique_ptr<EuropeanOption>{});
    m_options.back().swap(o);
}

double BasketOfOptions2::timeToExpiry() const {
    return m_options.at(0u)->timeToExpiry();
}

double BasketOfOptions2::payoffOnExpiry(double equityPrice) const {
    double p = 0.0;
    for(const auto& o : m_options){
        p += o->payoffOnExpiry(equityPrice);
    }
    return p;
}
```

I could demonstrate the functions by adding the following lines to the bottom of the `main` function in `main.cpp`.

```
BasketOfOptions basket;
basket.add(call);
basket.add(put);

BasketOfOptions2 basket2;
basket2.add(make_unique<EuropeanOption>(call));
basket2.add(make_unique<EuropeanOption>(put));
std::cout<<"\nSums:\n";
printResults(p.go(m,basket));
printResults(p.go(m,basket2));
```

(c) Implement a basket Instrument: create a new class `Basket` which owns several `Payoff`'s which all expire at the same time, and which implements the Payoff interface. Its payoff should be the sum of the payoffs of the options it contains. Note that you will have to add a virtual destructor to `Payoff` to make this work. Aside from adding the virtual destructor, this class will be the same as `BasketOfOptions2` but with `EuropeanOption` replaced by `Payoff`.

Ideally this class would have its own header file and implementation cpp file.