Introduction to Selected Classes of the QuantLib Library I

Dimitri Reiswich

December 2010

QuantLib is an open source library for Quantitative Finance. Some reasons to use it are

- It's free!
- If you're starting to code a C++ project from the scratch, you don't have to start from zero. This will allow you to focus on important projects, instead of spending time with coding basic classes such as Date, Interpolation or Yield Curve classes.
- You don't trust open source libraries? You have to admit that open source libraries have some advantages over commercial products. Here, the source code is available and you know exactly what happens in the background. No black-box computations. Clearly, the potential lack of documentation and user support is a disadvantage.
- You already have your own library? Fine. You can still use QuantLib to test and benchmark the results and performance of your own code.
- It is a good place to start learning advanced C++ concepts, such as the usage of design patterns.

In the whole tutorial I assume that you have included the QuantLib header via

```
#include <ql/quantlib.hpp>
```

in the code. I will not state this include command in the example code explicitly, so be sure that you have included the header everywhere. Also, I will use the QuantLib namespace to make the code more readable and compact. To reproduce the code, make sure to state

```
using namespace QuantLib;
```

in the place where you are using the code.

1. Useful Macros

2. Variable Types
   - Exercise

3. Date, Calendar Classes and Day Counters
   - Date
   - Calendars
   - Day Counters
   - Schedule
   - Exercise

4. Design Patterns
   - Factory
   - Singleton
   - Exercise
   - Observer, Observables
   - Quotes
   - Lazy Object
   - Handles

QuantLib has some useful macros, which will be introduced first. A few macros enable a compact error handling syntax. Imagine that you would like to check a variable against zero, before dividing. If the variable is zero, you'd like to throw an exception via a string. A common syntax to do this would be

```
if(x==0) {
    std::string myError("Divided by Zero");
    throw myError;
}
```

QuantLib handles the same event with a one-liner:

```
QL_REQUIRE(x!=0,"Divided by Zero")
```

The macro throws a `std::exception` which can be handled by the usual `catch` command. A similar macro is `QL_FAIL`, which accepts a message and no condition. Example code follows.

We will test the following functions

```
void testingMacros1(){
        double x=0.0;
        QL_REQUIRE(x!=0,"Zero number!");
}

void testingMacros2(){
        QL_FAIL("Failed!");
}
```

by calling them in a `try,catch` block as shown in the example below

```
int _tmain(int argc, _TCHAR* argv[])
{
        try{
                        testingMacros1();
        } catch (std::exception& e) {
                        std::cout << e.what() << std::endl;
                        return 1;
        } catch (...) {
                        std::cout << "unknown error" << std::endl;
                        return 1;
        }
}
```

The function `testingMacros1()` returns

```
Zero number!
```

while `testingMacros2()` returns

```
Failed!
```

You would like to see more output? For example, the name of the function where the error occurred? Or even the name of the file and the line? You can achieve this by changing the user configuration in

`<ql/userconfig.hpp>`

For the function name, change

```
#ifndef QL_ERROR_FUNCTIONS
// #define QL_ERROR_FUNCTIONS
#endif
```

to (uncomment the middle line)

```
#ifndef QL_ERROR_FUNCTIONS
#define QL_ERROR_FUNCTIONS
#endif
```

The same procedure can be used to show the file and line information by changing the `QL_ERROR_LINES` block in the same file. You need to recompile QuantLib to see the changes.

The `QL_LIB_VERSION` macro returns the current QuantLib version. For example

```
std::cout << "Current QL Version:" << QL_LIB_VERSION << std::endl;
```

yields

```
Current QL Version:0_9_7
```

QuantLib has various special variable types which aim at providing a more intuitive and clear code. The variables are `typedef` versions of the standard C++ variables such as `int`, `double`, `long`. For example, a function may return a variable of type `Volatility`. This is implicitly a `double` variable. Similarly, a variable of type `DiscountFactor` is provided.

Why all the extra variable types? Consider the following function

```
void myFunc(double df, double v, double r, double t)
```

This function accepts a discount factor, a volatility, an interest rate and the time to maturity. This is not obvious by looking at the variable names only, since all variables are of type `double`. Consider the QuantLib equivalent:

```
void myFunc(DiscountFactor df, Volatility v, Rate r, Time t)
```

Which one do you find more intuitive? If you are still not convinced: You can cast all variables easily to their original types.

The following variable types are available

- `Size` is a `std::size_t`
- `Integer` is an `int`
- `Natural` is an `int`
- `BigInteger` is a `long`
- `BigNatural` is a `long`
- `Real` is a `double`
- `Decimal` is a `double`
- `Time` is a `double`
- `DiscountFactor` is a `double`
- `Rate` is a `double`
- `Spread` is a `double`
- `Volatility` is a `double`

- As a warm up, write a function called `myReciproke` which returns $\frac{1}{x}$ for a variable x. The function should accept and return the numbers as a `Real` object. Check the variable x against zero before calculating and throw an error in this case. Catch the error in the main function. Check the functionality of your code.

A date in QuantLib can be constructed with the following syntax

```
Date(BigInteger serialNumber)
```

where `BigInteger` is the number of days such as 24214, and 0 corresponds to 31.12.1899. This date handling is also known from Excel. The alternative is the construction via

```
Date(Day d, Month m, Year y)
```

Here, `Day, Year` are of integer types, while `Month` is a special QuantLib type with

- January: either `QuantLib::January` or `QuantLib::Jan`
- ...
- December: either `QuantLib::December` or `QuantLib::Dec`

After constructing a Date, we can do simple date arithmetics, such as adding/subtracting days and months to the current date. Furthermore, the known convenient operators such as

$$++, --, +=, -=$$

can be used. Here, `++,--` add a single day to the existing date. It is possible to add a Period to a date with the `Period(int n,TimeUnit units)`, `TimeUnit` $\in$ `{Days, Weeks, Months, Years }` class. Example code is shown on the next slide.

```
void DateTesting1(){

        Date myDate(12,Aug,2009);
                std::cout << myDate << std::endl;
        myDate++;
                std::cout << myDate << std::endl;
        myDate+=12*Days;
                std::cout << myDate << std::endl;
        myDate-=2*Months;
                std::cout << myDate << std::endl;
        myDate--;
                std::cout << myDate << std::endl;


        Period myP(10, Weeks);
        myDate+=myP;
                std::cout << myDate << std::endl;
}
```

The output of the function is

```
August 12th, 2009
August 13th, 2009
August 25th, 2009
June 25th, 2009
June 24th, 2009
September 2nd, 2009
```

Each `Date` object has the following functions

- `weekday()` returns the weekday via the `QuantLib::Weekday` object
- `dayOfMonth()` returns the day of the month as an `QuantLib::Day` object
- `dayOfYear()` returns the day of the year as an `QuantLib::Day` object
- `month()` returns a `QuantLib::Month` object
- `year()` returns a `QuantLib::Year` object
- `serialNumber()` returns a `QuantLib::BigInteger` object

The function names should be self-explaining. Are you confused by all the new objects such as `QuantLib::Day` and `QuantLib::Year`? Do you ask yourself, if this is all necessary and if you couldn't have achieved the same via simple `int` objects? The good news is, that all objects can implicitly be converted to integers. For example, you can call `int myDay=myDate.dayOfYear()` instead of `Day myDay=dayOfYear()`, as illustrated in the code on the next slide.

The following code tests the introduced functions

```
void DateTesting2(){

        Date myDate(12,QuantLib::Aug,2009);

        std::cout << "Original Date:" << myDate << std::endl;
        std::cout << "Weekday:" << myDate.weekday() << std::endl;
        std::cout << "Day of Month:" << myDate.dayOfMonth() << std::endl;
        std::cout << "Day of Year:" << myDate.dayOfYear() << std::endl;
        std::cout << "Month:" << myDate.month() << std::endl;
                int month=myDate.month();
        std::cout << "Month via Integer:" << month << std::endl;
        std::cout << "Year:" << myDate.year() << std::endl;
                int serialNum=myDate.serialNumber();
        std::cout << "Serial Number:" << serialNum << std::endl;
}
```

The output of the previous code is

```
Original Date:August 12th, 2009
Weekday:Wednesday
Day of Month:12
Day of Year:224
Month:August
Month via Integer:8
Year:2009
Serial Number:40037
```

The QuantLib `Date` class has some useful `static` functions, which give general results, such as whether a given year is a leap year or a given date is the end of the month. The currently available functions are

- `Date::todaysDate()`
- `Date::minDate()`: earliest possible Date in QuantLib
- `Date::maxDate()`: latest possible Date in QuantLib
- `bool::isLeap(Year y)`: is `y` a leap year?
- `Date::endOfMonth(const Date& d)`: what is the end of the month, in which `d` is a day?
- `bool::isEndOfMonth(const Date& d)`: is `d` the end of the month?
- `Date::nextWeekday(const Date& d, Weekday w)`: on which date is the weekday `w` following the date `d`? (e.g. date of the next Friday)
- `Date nthWeekday(Size n, Weekday w,Month m,Year y)`:what is the n-th weekday in the given year and month? (e.g. date of the 3rd Wednesday in July 2010)

Example code for the introduced functions:

```
void DateTesting3(){
        std::cout << "Today:" << Date::todaysDate() << std::endl;
        std::cout << "Min Date:" << Date::minDate() << std::endl;
        std::cout << "Max Date:" << Date::maxDate() << std::endl;
        std::cout << "Is Leap:" << Date::isLeap(2011) << std::endl;
        std::cout << "End of Month:" << Date::endOfMonth(Date(4,Aug,2009)) << std::endl;
        std::cout << "Is Month End:" << Date::isEndOfMonth(Date(29,Sep,2009)) << std::endl;
        std::cout << "Is Month End:" << Date::isEndOfMonth(Date(30,Sep,2009)) << std::endl;
        std::cout << "Next WD:" << Date::nextWeekday(Date(1,Sep,2009),Friday) << std::endl;
        std::cout << "n-th WD:" << Date::nthWeekday(3,Wed,Sep,2009)<< std::endl;
}
```

The output of this function is

```
Today:September 1st, 2009
Min Date:January 1st, 1901
Max Date:December 31st, 2199
Is Leap :0
End of Month:August 31st, 2009
Is Month End:0
Is Month End:1
Next WD:September 4th, 2009
n-th WD:September 16th, 2009
```

Some situations may require the setting of the pricing date of an instrument to some past or future date. This instrument may depend on other instruments, which have to be evaluated at the new date too. To make such a procedure convenient, QuantLib provides a global settings instance, where you can set a global evaluation date. The current evaluation date can be retrieved via

```
Settings::instance().evaluationDate()
```

This date can be set to a customized evaluation date. Example code is shown below

```
void DateTesting4(){

        Date d = Settings::instance().evaluationDate();

        std::cout << "Eval Date:" << d << std::endl;
        Settings::instance().evaluationDate()=Date(5,Jan,1995);
        d=Settings::instance().evaluationDate();
        std::cout << "New Eval Date:" << d << std::endl;
}
```

The output of this function is

```
Eval Date:September 2nd, 2009
New Eval Date:January 5th, 1995
```

One of the crucial objects in the daily business is a calendar for different countries which shows the holidays, business days and weekends for the respective country. In QuantLib, a calendar can be set up easily via

```
Calendar myCal=UnitedKingdom()
```

for the UK. Various other calendars are available, for example for Germany, United States, Switzerland, Ukraine, Turkey, Japan, India, Canada and Australia. In addition, special exchange calendars can be initialized for several countries. For example, the Frankfurt Stock Exchange calendar can be initialized via

```
Calendar myCal=Germany(Germany::FrankfurtStockExchange);
```

The following functions are available:

- `bool isBusinessDay(const Date& d)`
- `bool isHoliday(const Date& d)`
- `bool isWeekend(Weekday w)`: is the given weekday part of the weekend?
- `bool isEndOfMonth(const Date& d)`: indicates, whether the given date is the last business day in the month.
- `Date endOfMonth(const Date& d)`: returns the last business day in the month.

The introduced functions are tested below

```
void CalendarTesting1(){

Calendar frankfCal=Germany(Germany::FrankfurtStockExchange);
Calendar saudiArabCal=SaudiArabia();
Date nyEve(31,Dec,2009);

std::cout << "Is BD:" << frankfCal.isBusinessDay(nyEve) << std::endl;
std::cout << "Is Holiday:" << frankfCal.isHoliday(nyEve) << std::endl;
std::cout << "Is Weekend:" << saudiArabCal.isWeekend(Saturday) << std::endl;
std::cout << "Is Last BD:" << frankfCal.isEndOfMonth(Date(30,Dec,2009)) << std::endl;
std::cout << "Last BD:" << frankfCal.endOfMonth(nyEve) << std::endl;

}
```

```
Is BD:0
Is Holiday:1
Is Weekend:0
Is Last BD:1
Last BD:December 30th, 2009
```

Note, that the Saturday is not a weekend in Saudi Arabia. As usual, the number 1 represents true while 0 represents false.

The calendars are customizable, so you can add and remove holidays in your calendar:

- `void addHoliday(const Date& d)`: adds a user specified holiday
- `void removeHoliday(const Date& d)`: removes a user specified holiday

Furthermore, a function is provided to return a vector of holidays

- `std::vector<Date> holidayList(const Calendar& calendar, const Date& from, const Date& to,bool includeWeekEnds )`: returns a holiday list, including or excluding weekends.

To continue the previous example from function `void CalendarTesting1()`, we can remove the 31-st of December as a holiday and add the 30-th, which was the last business day so far. Example code is given on the next slide.

```
#include <boost/foreach.hpp>

void CalendarTesting2(){

        Calendar frankfCal=Germany(Germany::FrankfurtStockExchange);
        Date d1(24,Dec,2009), d2(30,Dec,2009),d3(31,Dec,2009);

        frankfCal.addHoliday(d2);
        frankfCal.removeHoliday(d3);

        std::cout << "Is Business Day:" << frankfCal.isBusinessDay(d2) << std::endl;
        std::cout << "Is Business Day:" << frankfCal.isBusinessDay(d3) << std::endl;

        std::vector<Date> holidayVec=frankfCal.holidayList(frankfCal,d1,d2,false);
        std::cout << "----------" << std::endl;
        BOOST_FOREACH(Date d,holidayVec) std::cout << d << std::endl;
}
```

The output is

```
Is Business Day:0
Is Business Day:1
----------
December 24th, 2009
December 25th, 2009
December 30th, 2009
```

Adjusting a date can be necessary, whenever a transaction date falls on a date that is not a business day. The following Business Day Conventions are available in QuantLib:

- `Following`: the transaction date will be the first following day that is a business day.
- `ModifiedFollowing`: the transaction date will be the first following day that is a business day unless it is in the next month. In this case it will be the first preceding day that is a business day.
- `Preceding`: the transaction date will be the first preceding day that is a business day.
- `ModifiedPreceding`: the transaction date will be the first preceding day that is a business day, unless it is in the previous month. In this case it will be the first following day that is a business day.
- `Unadjusted`

The `QuantLib::Calendar` functions which perform the business day adjustments are

- `Date adjust(const Date&, BusinessDayConvention convention)`
- `Date advance(const Date& date,const Period& period, BusinessDayConvention convention,bool endOfMonth)`: the `endOfMonth` variable enforces the advanced date to be the end of the month if the current date is the end of the month.

Finally, it is possible to count the business days between two dates with the following function

- `BigInteger businessDaysBetween(const Date& from, const Date& to, bool includeFirst, bool includeLast)`: calculates the business days between `from` and `to` including or excluding the initial/final dates.

We will demonstrate an example by using the Frankfurt Stock Exchange calendar and the dates `Date(31,Oct,2009)` and `Date(1,Jan,2010)`. From the first date, we advance 2 months in the future, which is December, 31st. Since this is a holiday and the next business day is in the next month, we can check the Modified Following conversion. The Modified Preceding conversion can be checked for January, 1st 2010.

```
void CalendarTesting3(){

Calendar frankfCal=Germany(Germany::FrankfurtStockExchange);

        Date firstDate(31,Oct,2009);
        Date secondDate(1,Jan,2010);

std::cout <<"Date 2 Adv:" << frankfCal.adjust(secondDate,
        BusinessDayConvention(Preceding)) << std::endl;
std::cout <<"Date 2 Adv:" << frankfCal.adjust(secondDate,
        BusinessDayConvention(ModifiedPreceding)) << std::endl;

Period mat(2,Months);
std::cout << "Date 1 Month Adv:" << frankfCal.advance(firstDate,mat,
        BusinessDayConvention(Following),false) << std::endl;
std::cout << "Date 1 Month Adv:" << frankfCal.advance(firstDate,mat,
        BusinessDayConvention(ModifiedFollowing),false) << std::endl;

std::cout << "Business Days Between:" << frankfCal.businessDaysBetween(
        firstDate,secondDate,false,false) << std::endl;
}
```

The output of the function is

```
Date 2 Adv:December 30th, 2009
Date 2 Adv:January 4th, 2010
Date 1 Month Adv:January 4th, 2010
Date 1 Month Adv:December 30th, 2009
Business Days Between:41
```

Daycount conventions are crucial in financial markets. QuantLib offers

- `Actual360`: Actual/360 day count convention
- `Actual365Fixed`: Actual/365 (Fixed)
- `ActualActual`: Actual/Actual day count
- `Business252`: Business/252 day count convention
- `Thirty360`: 30/360 day count convention

The construction is easily performed via

```
DayCounter myCounter=ActualActual();
```

The other conventions can be constructed equivalently. The available functions are

- `BigInteger dayCount(const Date& d1, const Date& d2)`
- `Time yearFraction(const Date&, const Date&)`

Example code is provided on the next slide.

```
void dayCounterTesting1(){

        DayCounter dc=Thirty360();
        Date d1(1,Oct,2009);
        Date d2=d1+2*Months;

std::cout << "Days Between d1/d2:" <<dc.dayCount(d1,d2) << std::endl;
std::cout << "Year Fraction d1/d2:" <<dc.yearFraction(d1,d2) << std::endl;
}
```

The output of the function is

```
Days Between d1 and d2:60
Year Fraction d1 and d2:0.166667
```

An often needed functionality is a schedule of payments, for example for coupon payments of a bond. The task is to produce a series of dates from a start to an end date following a given frequency(e.g. annual, quarterly...). We might want the dates to follow a certain business day convention. And we might want the schedule to go backwards (e.g. start the frequency going backwards from the last date). For example:

- Today is `Date(3,Sep,2009)`. We need a monthly schedule which ends at `Date(15,Dec,2009)`. Going forwards would produce
  `Date(3,Sep,2009)`,`Date(3,Oct,2009)`,`Date(3,Nov,2009)`,`Date(3,Dec,2009)` and the final date `Date(15,Dec,2009)`.
- Going backwards, on a monthly basis, would produce
  `Date(3,Sep,2009)`,`Date(15,Sep,2009)`,`Date(15,Oct,2009)`, `Date(15,Nov,2009)`,`Date(15,Dec,2009)`.

The different procedures are given by the `DateGeneration` object and will now be summarized.

- `Backward`: Backward from termination date to effective date.
- `Forward`: Forward from effective date to termination date.
- `Zero`: No intermediate dates between effective date and termination date.
- `ThirdWednesday`: All dates but effective date and termination date are taken to be on the third Wednesday of their month (with forward calculation).
- `Twentieth`: All dates but the effective date are taken to be the twentieth of their month (used for CDS schedules in emerging markets). The termination date is also modified.
- `TwentiethIMM`: All dates but the effective date are taken to be the twentieth of an IMM month (used for CDS schedules). The termination date is also modified.

The schedule is initialized by the `Schedule` class, whose constructor is shown on the next slide.

```
Schedule(const Date& effectiveDate,
 const Date& terminationDate,
 const Period& tenor,
 const Calendar& calendar,
 BusinessDayConvention convention,
 BusinessDayConvention terminationDateConvention,
 DateGeneration::Rule rule,
 bool endOfMonth,
 const Date& firstDate = Date(),
 const Date& nextToLastDate = Date())
```

The variables represent the following

- `effectiveDate,terminationDate`: start/end of the schedule
- `tenor`: the frequency of the schedule (e.g. every 3 months)
- `terminationDateConvention`: allows to specify a special business day convention for the final date.
- `rule`: the generation rule, as previously discussed
- `endOfMonth`: if the effective date is the end of month, enforce the schedule dates to be end of the month too (termination date excluded).
- `firstDate,nextToLastDate`: are optional parameters. If we generate the schedule forwards, the schedule procedure will start from `firstDate` and then increase in the given periods from there. If `nextToLastDate` is set and we go backwards, the dates will be calculated relative to this date.

The `Schedule` object has various useful functions, we will discuss some of them.

- `Size size()`: returns the number of dates
- `const Date& operator[ ](Size i)`: returns the date at index `i`. Alternatively, there is the function `const Date& at(Size i)` to do the same thing.
- `Date previousDate(const Date& refDate)`: returns the previous date in the schedule compared to a reference date.
- `Date nextDate(const Date& refDate)`: returns the next date in the schedule compared to a reference date.
- `const std::vector<Date>& dates()`: returns the whole schedule in a vector.

The following function sets up a semi-annual schedule beginning on September, 30 th 2009 and ending on June, 15th 2012. We write the date vector and print each of the components.

```
#include <boost/foreach.hpp>

void testingSchedule1(){

        Date begin(30,September,2009), end(15,Jun,2012);
        Calendar myCal=Japan();

        BusinessDayConvention bdC=BusinessDayConvention(Following);

        Period myTenor(6,Months);

        DateGeneration::Rule myRule=DateGeneration::Forward;

        Schedule mySched(begin,end,myTenor,myCal,bdC,bdC,myRule,true);

        std::vector<Date> finalSched=mySched.dates();

        BOOST_FOREACH(Date d,finalSched) std::cout << d << std::endl;


}
```

The output of the function is:

```
September 30th, 2009
March 31st, 2010
September 30th, 2010
March 31st, 2011
September 30th, 2011
March 30th, 2012
June 15th, 2012
```

Note, how the scheduled dates were shifted to the end of the respective month. Switching the `rule` from `Forward` to `Backward` in the same program yields

```
September 30th, 2009
December 15th, 2009
June 15th, 2010
December 15th, 2010
June 15th, 2011
December 15th, 2011
June 15th, 2012
```

Now, we test the next and previous date functions. Suppose today's date is August, 3rd 2010. We would like to know the next and previous date in the example schedule, relative to today. The code and output is given below

```
void testingSchedule3(){

        Date begin(30,September,2009), end(15,Jun,2012);
        Calendar myCal=Japan();

        BusinessDayConvention bdC=BusinessDayConvention(Following);

        Period myTenor(6,Months);
        DateGeneration::Rule myRule=DateGeneration::Forward;

        Schedule mySched(begin,end,myTenor,myCal,bdC,bdC,myRule,true);

        Date myDate(3,Aug,2010);
        std::cout << "Date:" << myDate << std::endl;
        std::cout << "Next Date:" << mySched.nextDate(myDate) << std::endl;
        std::cout << "Prev Date:" << mySched.previousDate(myDate) << std::endl;

}
```

```
Date:August 3rd, 2010
Next Date:September 30th, 2010
Prev Date:March 31st, 2010
```

The constructor of the `Schedule` class is quite complex. QuantLib provides a `MakeSchedule` class which generates a schedule more conveniently. The basic structure of the syntax is as follows:

```
Schedule mySchedule=MakeSchedule(effectiveDate, terminationDate,
tenor,calendar,convention).function1().function2()....functionLast();
```

This is an example of the factory pattern: the `MakeSchedule` class is the factory producing a `Schedule` object. The functions can be called in any order such that you do not have to remember in which order the variables are initialized. An example for a function is `withRule(DateGeneration::Rule)`. An example setup with `MakeSchedule` is shown next.

```
void testingSchedule4(){

        Date begin(30,September,2009), end(15,Jun,2012);
        Calendar myCal=Japan();

        BusinessDayConvention bdC=BusinessDayConvention(Following);
        Period myTenor(6,Months);

    Schedule myScheduleMade=MakeSchedule(begin,end,myTenor,myCal,bdC)
                .backwards()
                .endOfMonth(false)
                .withNextToLastDate(Date(24,Aug,2011));

        std::vector<Date> finalSched=myScheduleMade.dates();
        BOOST_FOREACH(Date d,finalSched) std::cout << d << std::endl;


}
```

The output is

```
September 30th, 2009
February 24th, 2010
August 24th, 2010
February 24th, 2011
August 24th, 2011
June 15th, 2012
```

- Find out which of the following countries has the maximum number of holidays in 2009 (weekends excluded): Germany, UK, United States, Switzerland, Japan.
- Choose the calendar of the respective country and calculate the maximum number of business days between two holidays (e.g. the maximum number of days you have to work until the next holiday)

QuantLib uses factory classes within its code. Although no explicit class template is provided, we will show how this pattern is realized in QuantLib. This will make it easier for the user to understand some classes and is a nice code design example. Consider the `MyOption` class below

```
class MyOption{
public:
        enum Type{Call=1,Put=-1,None=0};

        MyOption(Time mat, Volatility vol, Type type, Real spot,
                Rate forRate, Rate domRate, Real strike):mat_(mat),vol_(vol),
                type_(type),spot_(spot),forRate_(forRate),domRate_(domRate),
                strike_(strike){
        }

        Time getMat()const{return mat_;}
        Real getSpot()const{return spot_;}
        Real getStrike()const{return strike_;}

private:
        Time mat_; Volatility vol_;
        Real spot_,strike_;
        Rate forRate_,domRate_;
        Type type_;
};
```

Despite being relatively simple, the class needs a lot of input variables. A user needs to know the exact order of the input variables at the time of an instance setup (although some IDE tools are available to provide assistance here). Also, it can be useful to provide default parameters to some variables such as `type_=MyOption::Call`. This is possible without any specific design by simply setting `type_=MyOption::Call` in the constructor. However, this requires a reordering of the variables as default variables have to appear at the end of the constructor.

In `QuantLib` you will often find classes which are characterized by a `Make` attribute in their name (i.e. `MakeSwaption`). In our illustrative case we will call the class `MakeMyOption`. This class can have any default constructor which doesn't need to initialize all variables. The member variables will be initialized by factory functions. The typical syntax of such a function is

```
MakeMyOption& withVol(const Volatility& vol){
vol_=vol;
return *this;
}
```

Such a function overwrites the respective member variable and returns the current object **by reference**. Invoking another function on the new object will overwrite additional variables.

This syntax allows a consecutive calling of many factory functions with a convenient syntax. The order of the functions is not relevant, you do not have to remember if volatility or the interest rate is written first. The last point is the extraction of the finished `MyOption` instance from the `MakeMyOption` factory class. This is achieved by a cast operator of the form

- `operator MyOption(){.... return option_;}`

The cast operator is familiar from cast operations like `int i=2; double x=(double)i`, which casts a variable of type `int` to a variable of type `double`. An equivalent syntax would be `double x=i;`. In `C++` it is possible to define an own cast operator with the syntax shown above. Consequently, it will be possible to write

- `MyOption option=MakeMyOption();`

The cast operator of `MakeMyOption` returns a built `MyOption` instance, it is a factory for the `MyOption` class. Although being very convenient, the setup bears some risks. The programmer has to decide what to do if some of the variables is not initialized. He can either throw an exception or give the variable a default value. The first solution is safer but aborts the program if the exception is not handled appropriately. The second solution can be risky because the user does not know that he forgot to initialize some variables. He will then use some default value without being aware of it. In our example we choose the first version and initialize all variables with `Null<Real>` when the default constructor is called. In the cast operator, all variables are checked against this value and an exception is thrown if the variable was not initialized. The final `MakeMyOption` class code is shown below.

```
class MakeMyOption{
public:

        MakeMyOption(): mat_(Null<Real>()),vol_(Null<Real>()),spot_(Null<Real>()),
                                    strike_(Null<Real>()),forRate_(Null<Real>()),
                                    domRate_(Null<Real>()),type_(MyOption::None){
        }
        MakeMyOption& withMat(const double& mat){
                mat_=mat;
                return *this;
        }
        MakeMyOption& withVol(const double& vol){
                vol_=vol;
                return *this;
        }
        MakeMyOption& withSpot(const double& spot){
                spot_=spot;
                return *this;
        }
        MakeMyOption& withStrike(const double& strike){
                strike_=strike;
                return *this;
        }
        MakeMyOption& withForRate(const double& forRate){
                forRate_=forRate;
                return *this;
        }
        MakeMyOption& withDomRate(const double& domRate){
                domRate_=domRate;
                return *this;
        }
        MakeMyOption& withType(const MyOption::Type& type){
                type_=type;
                return *this;
        }
        operator MyOption()const{
                QL_REQUIRE(mat_!=Null<Real>(),"Maturity not set!");
                // Check other parameters too
                QL_REQUIRE(type_!=MyOption::None,"Option type not set!");
                return MyOption(mat_,vol_,type_,spot_,forRate_,domRate_,strike_);
        }

        Time getMat()const{return mat_;}
```

An example initialization is shown below

```
void testingFactory2(){

        Real spot=100.0, strike=110.0;
        Rate rd=0.03, rf=0.01;
        Volatility vol=0.20;
        Time mat=1.0;
        MyOption::Type type(MyOption::Call);

        MyOption optionMade=MakeMyOption()
                        .withType(type)
                        .withMat(mat)
                        .withSpot(spot)
                        .withForRate(rf)
                        .withStrike(strike)
                        .withVol(vol)
                        .withDomRate(rd);

        std::cout << "Mat Made:"    << optionMade.getMat() << std::endl;
        std::cout << "Spot Made:"       << optionMade.getSpot() << std::endl;
        std::cout << "Strike Made:"     << optionMade.getStrike() << std::endl;

}
```

Using the factory design yields a more readable code, which is also easier to setup. Also, the order of the initialization is irrelevant. The output of the function is

```
Mat Made:1
Spot Made:100
Strike Made:110
```

The Singleton pattern is a design where a class is allowed to have only one instance. For example, one application is a global repository where you store market objects. In this case it is important to have a single repository that is used by all classes, e.g. you want the pricing to be based on the same yield curve. A global variable does not meet these requirements, since it can not be ensured that multiple objects are instantiated. To implement a QuantLib Singleton object you need to derive from the `Singleton` in the following exemplary way

```
class Foo : public Singleton<Foo> {
            friend class Singleton<Foo>;
          private:
            Foo() {}
          public:
            ...
        };
```

An instance of the class is retrieved by the `::instance()` function. We will demonstrate the usage by a heterogeneous global repository example.

The repository has a function called

```
void addObject(const std::string& id, const boost::any& obj, bool overwrite = true)
```

which adds an object with its respective ID to the repository. Note that you don't have to pass an `any` object but "any" object. The boolean indicates if any object with the same name shall be overwritten or not. The object can be retrieved with

```
template <class T> boost::optional<T> getObject(const std::string& id, Error& err)
```

The error variable is optional and writes an error in case the object could not be found (either due to a non existing ID or due to a casting failure). An object of type `optional` is returned. Before proceeding one can check if this is a 0 (failed) or 1 (success) variable and dereference it afterwards. Additional functions are

- `bool objectExists(const std::string& id)`
- `void deleteObject(const std::string& id)`
- `void deleteAllObjects()`
- `unsigned int numberObjects()` returns the total number of objects
- `template <class T> unsigned int getObjectCount()` get number of objects with type `T`

```
#include <boost/optional.hpp>

#ifndef object_repository_h
#define object_repository_h

class ObjectRepository: public QuantLib::Singleton<ObjectRepository>{

        friend class QuantLib::Singleton<ObjectRepository>;

public:

        enum Error{NoError,CastingFailed,IdNotFound};
        // function to add objects

        // ------------------------------------------------
        // add and delete objects
        void addObject(const std::string& id, const boost::any& obj, bool overwrite = true){

                it_=(*rep_).find(id);

                if(it_!=(*rep_).end()){

                        if(overwrite==true){
                                (*rep_)[id]=obj;
                        }
                        else{
                                QL_FAIL("Can not overwrite object.");
                        }
                }
                else{
                        (*rep_)[id]=obj;
                }

        }

        void deleteObject(const std::string& id){

                it_=(*rep_).find(id);

                if(it_!=rep_->end()){
                        rep_->erase(it_);
                }
        }
```

```
#include <boost/foreach.hpp>
#include "Singleton1.h"

void addVariable();

void testingSingleton1(){

        addVariable();

        std::string myStr1("myStr1");
        std::vector<int> myVec1(6,2);

        ObjectRepository::instance().addObject("str1",myStr1);
        ObjectRepository::instance().addObject("vec1",myVec1);

        std::string myDblId("dbl1");

        ObjectRepository::Error myErr;

        boost::optional<double> myDblGet=ObjectRepository::instance().getObject<double>(myDblId,myErr);

        std::cout << "Object exits?: " << ObjectRepository::instance().objectExists(myDblId)<<std::endl;
        std::cout << "getObject double return: "<< myDblGet << std::endl;
        std::cout << "dereferenced double return: "<< *myDblGet << std::endl;
        std::cout << "Error: " << myErr << std::endl;
        std::cout << "--------------------------------" << std::endl;

        boost::optional<std::vector<int>> myVecGet=
                                ObjectRepository::instance().getObject<std::vector<int>>("vec1");

        std::cout << "getObject vector return: " << myVecGet << std::endl;

        BOOST_FOREACH(int x,*myVecGet) std::cout << x << std::endl;

}

void addVariable(){
        double myDbl1=2.123;
        ObjectRepository::instance().addObject("dbl1",myDbl1);
}
```

The output of the function is

```
Object exits?:  1
getObject double return:  1
dereferenced double return:  2.123
Error:  NoError
------------------------------
getObject vector return:  1
2
2
2
2
2
2
```

A function testing the other functions is shown below

```
#include <boost/foreach.hpp>
#include "Singleton1.h"

void addVariable();

void testingSingleton2(){

        try{

        addVariable();

        double myDbl2=4.144,myDbl3=3.122;
        std::string myStr1("myStr1"),myStr2("myStr2");
        std::vector<int> myVec1(6,2),myVec2(10,1),myVec3(2,2),myVec4(4,1);

        ObjectRepository::instance().addObject("dbl2",myDbl2);
        ObjectRepository::instance().addObject("dbl3",myDbl3);


        ObjectRepository::instance().addObject("str1",myStr1);
        ObjectRepository::instance().addObject("str2",myStr2);

        ObjectRepository::instance().addObject("vec1",myVec1);
        ObjectRepository::instance().addObject("vec2",myVec2);
        ObjectRepository::instance().addObject("vec3",myVec3);
        ObjectRepository::instance().addObject("vec4",myVec4);

        std::cout << "Number Objects:" << ObjectRepository::instance().numberObjects() << std::endl;
        std::cout << "Number Doubles:" << ObjectRepository::instance().getObjectCount<double>() << std::endl;
        std::cout << "Number Strings:" << ObjectRepository::instance().getObjectCount<std::string>() << std::end
        std::cout << "Number Vectors:" << ObjectRepository::instance().getObjectCount<std::vector<int>>() << std
        std::cout << "--------------------------------" << std::endl;

        std::string myDblId("dbl1");
        ObjectRepository::Error err;

        boost::optional<double> myDblGet =ObjectRepository::instance().getObject<double>(myDblId,err);

        std::cout << "Object exits?: " << ObjectRepository::instance().objectExists(myDblId)<<std::endl;
        std::cout << "getObject double return: "<< myDblGet << std::endl;
        std::cout << "dereferenced double return: "<< *myDblGet << std::endl;
```

```
Number Objects:9
Number Doubles:3
Number Strings:2
Number Vectors:4
-------------------------------
Object exits?:  1
getObject double return:  1
dereferenced double return:  2.123
Error:  NoError
-------------------------------
Object exits?:  0
getObject after delete return:  0
Error:  IdNotFound
-------------------------------
getObject wrong type return:  0
Error:  CastingFailed
-------------------------------
getObject wrong id return:  0
Error:  IdNotFound
-------------------------------
getObject vector return:  1
Error:  NoError
2
2
2
2
2
```

- Write 2 void sub-functions `addOption1()` and `addOption2()` which store 2 different `MyOption` objects(created with the factory pattern) in the repository. Call the functions in a main function to store the objects, try to extract them afterwards and check if the option properties are recovered correctly.

One of the most important design patterns are the `Observer, Observable` patterns. This pattern defines a one-to-many relationship, such that the change of an object-state is updated automatically in all dependent objects. One can think of many examples in Quantitative Finance where this is needed. For example, think about a yield curve which is passed as an object to other classes. Since yields change on a regular basis, you would like to update the dependent classes such that the pricing is based on an updated curve. This is exactly the purpose of QuantLib's Observer/Observable patterns. The `Observer` class has a default and a copy constructor without any input arguments. The functionality is added by two public non-virtual functions called

```
void registerWith(const boost::shared_ptr<Observable>&);
void unregisterWith(const boost::shared_ptr<Observable>&);
```

which implement the registration policy for Observables. Note, that a `boost::shared_ptr` to an `Observable` has to be passed. Furthermore, a public, pure `virtual` update function is given with

```
virtual void update()
```

This function is called by the observed objects whenever they change, and needs to be overwritten. To use the `Observer` class, you need to derive from this class and implement the update function.

The `Observable` class has a default and copy constructor and a public function called

```
void notifyObservers();
```

The user needs to call the function at a suitable place in classes which are derived from the `Observable` class. It remains to decide when and whether the observers should be notified. Calling this function triggers the `update()` function of all observers. Note, that an `Observable` can be an `Observer` at the same time.

To illustrate the pattern, let us implement a simple yield and discount factor classes. The yield class is an `Observable` which changes on a regular basis. The discount factor class is an `Observer` with a yield class instance as a member. It returns the discount factor for a given time. The yield class will have a `void setYield(...)` function, which notifies the observers since the yield has changed. The discount factor will have an `update()` function which recalculates the discount factor based on the new yield. The discount factor class will be an `Observable` at the same time in case other classes need to observe it. The code setup is illustrated next.

```
class SimpleYield: public Observable{
private:
        Rate yield_;
public:
        SimpleYield(const Rate& yield):yield_(yield){
        }
        Rate getYield() const{return yield_;}

        void setYield(const Rate& yield){
                yield_=yield;
                // yield has changed, notify observers!
                notifyObservers();
        }
};
class SimpleDiscountFactor: public Observable, Observer{
private:
        DiscountFactor df_;
        Time mat_;
        boost::shared_ptr<SimpleYield> y_;
public:
        SimpleDiscountFactor(const boost::shared_ptr<SimpleYield>& y,
                             Time& mat):y_(y),mat_(mat){
                             // register yield as an observable!
                             registerWith(y_);
                             df_=exp(-y_->getYield()*mat_);
        }
        void update(){
                // something has changed, recalculate yield
                df_=exp(-y_->getYield()*mat_);
                notifyObservers();
        }
        Real getDiscount() const{
                return df_;
        }
};
```

We will test the code by setting up a yield and discount factor class and changing the yield with the `setYield(...)` function.

```
void testingDesignPatterns2(){

        boost::shared_ptr<SimpleYield> myYield(new SimpleYield(0.03));
        Time mat=1.0;
        SimpleDiscountFactor myDf(myYield, mat);
        std::cout << "Discount before update:" << myDf.getDiscount() << std::endl;
        myYield->setYield(0.01);
        std::cout << "Discount after update:" << myDf.getDiscount() << std::endl;

}
```

The output of this function is

```
Discount before update:0.970446
Discount after update:0.99005
```

As the result shows, the discount factor instance returns an updated factor without changing anything in the instance directly.

We will take a further step in the previous example and illustrate how the discount factor class can be observed by other classes (remember that it has been derived from the `Observable` class too). We will define a discounted cash flow class `SimpleDiscountedCF`, which discounts any cash flow given a discount factor. This class needs an updated discount factor. Changing the yield in the original yield class should trigger the following procedures

- Update the discount factor class, since yield has changed. Notify the observer of the discount factor class (the `SimpleDiscountedCF` class in this case).
- Update the discounted cash flow class, since the discount factor has changed.

The example class is shown below

```
class SimpleDiscountedCF:public Observer{
private:
        boost::shared_ptr<SimpleDiscountFactor> df_;
        Real discountedUnit_;
public:
        SimpleDiscountedCF(const boost::shared_ptr<SimpleDiscountFactor>& df):df_(df){
                        discountedUnit_=df_->getDiscount();
                        registerWith(df_);
        }
        void update(){
                // something has changed, recalculate discount factor
                discountedUnit_=df_->getDiscount();
        }
        Real discountCashFlow(const Real& amount) const{
                return discountedUnit_*amount;
        }
};
```

The functionality is tested with a function which will discount a cash flow of 100.0. We will test the discounted cash flow before and after updating the initialized single yield.

```
void testingDesignPatterns3(){

        boost::shared_ptr<SimpleYield> myYield(new SimpleYield(0.03));
        Time mat=1.0;
        // construct observer of yield curve
        boost::shared_ptr<SimpleDiscountFactor> myDf(
                new SimpleDiscountFactor(myYield, mat));
        // construct observer of discount factor
        SimpleDiscountedCF myCf(myDf);

        std::cout << "Cash Flow before update:" << myCf.discountCashFlow(100.0) << std::endl;
        myYield->setYield(0.01);
        std::cout << "Cash Flow after update:" << myCf.discountCashFlow(100.0) << std::endl;

}
```

The corresponding output is

```
Cash Flow before update:97.0446
Cash Flow after update:99.005
```

Before proceeding to the next topic, we will introduce an interesting `Observable` which is the `evaluationDate()` in the global `Settings` class. Recalling, that the date can be called by

```
Settings::instance().evaluationDate()
```

it is possible to register with the evaluation date via

```
registerWith(Settings::instance().evaluationDate())
```

The corresponding `Observer` will be notified any time the evaluation date changes, which makes it possible to enforce a revaluation of all instruments by changing the global evaluation date. This is obviously something that is needed for all financial instruments which have a time to maturity.

To demonstrate the functionality, we will rewrite the previously discussed `SimpleDiscountFactor` class to a class which accepts a maturity date and day counter, instead of a time to maturity. This class will be called `SimpleDiscountFactor1` class. The valuation date will be the global one, which we will register as an `Observable`. The `SimpleDiscountFactor1` class will be notified in case the global evaluation date changes, which will trigger the `update()` function. This function will then recalculate the discount factor based on the new evaluation date (and time to maturity). The class code is shown below

```
class SimpleDiscountFactor1: public Observable, Observer{
private:
        DiscountFactor df_;
        Date evalDate_,matDate_;
        boost::shared_ptr<SimpleYield> y_;
        DayCounter dc_;
public:
        SimpleDiscountFactor1(const boost::shared_ptr<SimpleYield>& y,
                              const Date& matDate, const DayCounter& dc)
                              :y_(y),matDate_(matDate),dc_(dc){
                              // register yield as an observable!
                              evalDate_=Settings::instance().evaluationDate();
                              registerWith(y_);
                              registerWith(Settings::instance().evaluationDate());
                              df_=exp(-y_->getYield()*dc_.yearFraction(evalDate_,matDate_));
        }
        void update(){
                // something has changed, recalculate discount factor
                evalDate_=Settings::instance().evaluationDate();
                df_=exp(-y_->getYield()*dc_.yearFraction(evalDate_,matDate_));
                notifyObservers();
        }
        Real getDiscount() const{
                return df_;
        }
};
```

To test the registration, we will change the yield and afterwards the global evaluation date and print out the corresponding result. This is summarized in the following function

```
void testingDesignPatterns2a(){

        boost::shared_ptr<SimpleYield> myYield(new SimpleYield(0.03));
        Date mat=Date::todaysDate()+12*Months;
        DayCounter dc=ActualActual();

        SimpleDiscountFactor1 myDf(myYield, mat,dc);

        std::cout << "Discount before yield update:" << myDf.getDiscount() << std::endl;
        myYield->setYield(0.01);
        std::cout << "Discount after yield update:" << myDf.getDiscount() << std::endl;
        Settings::instance().evaluationDate()=mat-1*Months;
        std::cout << "Discount after evaluation date update:" << myDf.getDiscount() << std::endl;

}
```

The output of the function is

```
Discount before yield update:0.970446
Discount after yield update:0.99005
Discount after evaluation date update:0.999151
```

The `Observer, Observable` patterns are very convenient to keep track of some market quotes, such as a volatility quote, a yield or a forward rate. `QuantLib` provides a base class called `Quote` from which one can derive any customized quote. The class implements two pure virtual functions called

- `Real value() const`
- `bool isValid()`

which return either the value or a boolean indicating whether the quote is valid. The `Quote` class is by default an `Observable`. A class that derives from the `Quote` class is `SimpleQuote`, which accepts a `Real` number in the constructor. In addition to the `Quote` functions, it implements a

- `Real setValue(Real value)` function, which sets the new value, notifies all observers and returns the difference to the old value.
- `void reset()` function which resets the quote changing it to an `isValid()=false` state.

Other classes derived from `Quote` are available, such as

- `LastFixingQuote`
- `ImpliedStdDevQuote`
- `ForwardValueQuote`
- `ForwardSwapQuote`
- `EurodollarFuturesImpliedStdDevQuote`
- `FuturesConvAdjustmentQuote`

All quotes implement some market specific properties and are an `Observer` as well as an `Obserable`.

This section discusses the `LazyObject` class which implements the lazy object design pattern. As the name suggests, the class is designed to be lazy with respect to any calculations. The classical case is the following: recall the observer/observable case where the `Observable` triggers the `update()` function of the `Observer`. This is a nice feature. However, the same feature can lead to unnecessary calculations. Imagine a volatility smile class which accepts 10 volatilities for some given strikes. Each volatility is an `Observable`, as is the whole volatility smile class. Now imagine what happens in case of a parallel shift of the smile:

- the first strike volatility changes, notifies the smile which again notifies its observers, which do some recalculations in the `update()` function.
- the second volatility changes, notifies the smile, the smile notifies its observers which do again some recalculations.
- ...

At the end we would have 10 recalculations, although we are actually interested in one recalculation after all volatilities are updated. One example is a recalibration of a model fitted to market quotes. Such a recalibration can be numerically very intensive and the goal should be one recalibration after all quotes are up to date and not a recalibration after each single volatility movement.

The `LazyObject` implements a design pattern which allows us to conveniently adjust for such a case. The class inherits from both, the `Observer` and `Observable` class. However, the user can not modify the `update()` function. This function is implemented by default and the only thing that it does is setting a `bool calculated_` to false and notifying all `Observers`. You can even avoid the notification of the `Observers` by using the `freeze()` function. This effect can be reverted by the `unfreeze()` function.

The class has one `protected` pure virtual function which needs to be implemented, called `void performCalculations()`. This function needs to be implemented once in each derived class. It is not designed to be called directly in any of the other functions of this class. The main function which should be used for the updating mechanism in the derived class is the `virtual void calculate()` function which is `protected`. The function does the following

- If the class is not frozen and the `calculated_` flag is set to false, it tries to call `void performCalculations()` and sets `calculated_` to true. Otherwise it doesn't do anything.

The `void calculate()` is the function which should be called in other member functions before returning any value. If all stored calculations are up to date, the calling of `calculate()` doesn't have any effect, except from testing a simple `if` condition. Otherwise, it will call `performCalculations()` once and proceed with the updated calculated values.

We will demonstrate the `LazyObject` pattern for a SABR model calibration. The interpolation toolbox of `QuantLib` provides a `SABRInterpolation`. This model has been introduced in Hagan et al. 2002. It's name is due to the α, β, ρ coefficients used in the following SDE system

$$dF_t = \hat{\alpha}_t F_t^{\beta} dW_t^1 \qquad F_0 = f \tag{1}$$

$$d\hat{\alpha}_t = \nu \hat{\alpha}_t dW_t^2. \qquad \hat{\alpha}_0 = \alpha. \tag{2}$$

Here, F_t is the stochastic process describing the evolution of the forward of an underlying asset and $\hat{\alpha}_t$ is a stochastic scaling parameter. The Wiener processes W_t^1, W_t^2 are correlated via $dW_t^1 dW_t^2 = \rho dt$. Hagan et al. derive an approximative formula for the implied Black-Scholes volatility, which is implied by the model. This function depends on the parameters α, β, ρ, ν and is given as

$$\begin{aligned}\sigma_{SABR}(K, \alpha, \beta, \rho, \nu) &= \frac{\alpha}{(fK)^{0.5(1-\beta)}\left[1 + \frac{(1-\beta)^2}{24}\ln^2\frac{f}{K} + \frac{(1-\beta)^4}{1920}\ln^4\frac{f}{K}\right]}\left(\frac{z}{\chi(z)}\right) \\ &\times \left[1 + \left(\frac{(1-\beta)^2}{24}\frac{\alpha^2}{(fK)^{1-\beta}} + \frac{1}{4}\frac{\rho\beta\nu\alpha}{(fK)^{0.5(1-\beta)}} + \frac{2-3\rho^2}{24}\nu^2\right)\tau\right]\end{aligned}$$

with

$$z = \frac{\nu}{\alpha}(fK)^{0.5(1-\beta)} \ln\frac{f}{K}$$

and

$$\chi(z) = \ln\left(\frac{\sqrt{1-2\rho z + z^2} + z - \rho}{1-\rho}\right).$$

Given a market strike-volatility function, the objective in the calibration is to find parameters α, β, ρ, ν which fit the market smile. This requires a calibration, a multidimensional optimization algorithm with an appropriate cost function. This is incorporated in the constructor of the interpolation

```
SABRInterpolation(const I1& xBegin,  // x = strikes
                  const I1& xEnd,
                  const I2& yBegin,  // y = volatilities
                  Time t,            // option expiry
                  const Real& forward,
                  Real alpha,
                  Real beta,
                  Real nu,
                  Real rho,
                  bool alphaIsFixed,
                  bool betaIsFixed,
                  bool nuIsFixed,
                  bool rhoIsFixed,
                  bool vegaWeighted = false,
                  const boost::shared_ptr<EndCriteria>& endCriteria
                        = boost::shared_ptr<EndCriteria>(),
                  const boost::shared_ptr<OptimizationMethod>& optMethod
                        = boost::shared_ptr<OptimizationMethod>())
```

The interpolation constructor allows to decide whether some of the parameters should be kept fix (a popular choice is $\beta = 0.7$ or $\beta = 1.0$).

Imagine that we have a `LazyObject SabrModel` class which observes an instance of a `SimpleSmile` class and does not have any observers. The `SimpleSmile` class is constructed as an `Observer` of several volatility vs. strike quotes. We are interested in the procedures which are called after `SimpleSmile` is updated by any of the volatility quotes. If this happens, the `SimpleSmile` class triggers the `update()` function of `SabrModel`, which doesn't trigger any calculations(`notifyObservers()` will have no effect). The only thing that `update()` does is setting the `calculated_` flag to `false`. Since the `SabrModel` class has a recalibration procedure, we will put it into the `void performCalculations()` function.

Another function that `SabrModel` will have is a

```
getVanillaPrice(const Real& strike)
```

function. Clearly, we want this function to return the vanilla option price based on a calibrated parameter set. To do this, we will call `calculate()` at the beginning of the function. If no calculations are necessary (`calculated_` is set to `true`) `performCalculations()` will not be called and no recalibration is performed. Otherwise, a recalibration is invoked first before proceeding to the vanilla price calculation. The concepts will be presented in some simple classes.

The `SimpleVolQuote` and `SimpleSmile` class are presented below

```
class SimpleVolQuote: public Quote{
private:
        Volatility vol_;
        Real strike_;

public:
        SimpleVolQuote(Volatility vol, Real strike) :vol_(vol),strike_(strike){}

        bool isValid() const {return vol_!=Null<Real>();}
        Real value() const {return vol_;}
        Real strike()const {return strike_;}

        void setVolatility(const Volatility& vol){
                vol_=vol;
                notifyObservers();
        }
};
class SimpleSmile: public Observable, Observer{
private:
        std::vector<boost::shared_ptr<SimpleVolQuote>> volVec_;
public:
        SimpleSmile(const std::vector<boost::shared_ptr<SimpleVolQuote>>& volVec)
                :volVec_(volVec){
                for(Size i=0;i<volVec_.size();i++){
                        registerWith(volVec_[i]);
                }
        }
        void update(){
                notifyObservers();
        }

        std::vector<boost::shared_ptr<SimpleVolQuote>> getVolVec() const{return volVec_;}
        Size getVolNumber() const{return volVec_.size();}
};
```

We will use the following simple Black-Scholes function for our calculations

```
#include<boost/math/distributions.hpp>

Real blackScholesPriceFwd(const Real& fwd,
                                                    const Real& strike,
                                                    const Volatility& vol,
                                                    const Rate& rd,
                                                    const Rate& rf,
                                                    const Time& tau,
                                                    const Integer& phi){
        boost::math::normal_distribution<> d(0.0,1.0);
        Real dp,dm, stdDev, res, domDf, forDf;

        domDf=std::exp(-rd*tau); forDf=std::exp(-rf*tau);
        stdDev=vol*std::sqrt(tau);

        dp=(std::log(fwd/strike)+0.5*stdDev*stdDev)/stdDev;
        dm=(std::log(fwd/strike)-0.5*stdDev*stdDev)/stdDev;

        res=phi*domDf*(fwd*cdf(d,phi*dp)-strike*cdf(d,phi*dm));
        return res;
}
```

```
#include "LazyObject1.h"
#include "LazyObjectBS.h"

class SabrModel: public LazyObject{

public:
        SabrModel(const boost::shared_ptr<SimpleSmile>& smile,
                const Real& fwd, const Time& tau, const Real& rd,const Real& rf)
                :smile_(smile),fwd_(fwd),tau_(tau),rd_(rd),rf_(rf),
                strikeVec_      (std::vector<Real>(smile->getVolNumber())),
                volVec_         (std::vector<Real>(smile->getVolNumber())){
                        // register smile as observable
                        registerWith(smile_);
        }

        Real getVanillaPrice(const Real& strike){
                calculate();
                return blackScholesPriceFwd(fwd_,strike,(*intp_)(strike),rd_,rf_,tau_,1);
        }

private:

        void performCalculations() const{
                volQuoteVec_=smile_->getVolVec();
                for(Size i=0;i< volQuoteVec_.size();++i){
                        strikeVec_[i]=(*volQuoteVec_[i]).strike();
                        volVec_[i]=(*volQuoteVec_[i]).value();
                }

                if(intp_==NULL){
                        intp_.reset(new SABRInterpolation(strikeVec_.begin(), strikeVec_.end(),
                        volVec_.begin(),tau_,fwd_,0.1,0.1,0.1,0.1,false, false, false, false));
                }
                        intp_->update(); std::cout << "Recalibration Performed!" << std::endl;

        }

        Real fwd_,rd_,rf_;
        Time tau_;
        boost::shared_ptr<SimpleSmile> smile_;
        mutable boost::shared_ptr<SABRInterpolation> intp_;
        mutable std::vector<Real> strikeVec_,volVec_;
```

The constructor does not perform any initial calibration, this is totally handled by the `calculate()` function. Before returning the price, the `getVanillaPrice` function calls the `calculate()` function which calls `performCalculations()` only if necessary. The `performCalculations()` function initializes the interpolation, if necessary, and updates the volatilities and strike vectors by performing a recalibration. This is the numerically intensive function. We have incorporated a print out of

```
"Recalibration Performed !"
```

in case the recalibration is called. This will be needed for the following test setup. We will construct 4 strike volatility quotes which will be used to construct an instance of `SimpleSmile`. Using this instance, an instance of the `SabrModel` class will be constructed. In addition, the following will be performed:

- Calculate 4 vanilla prices with strikes $90, 95, 100, 105$
- Shift all volatilities by 2%
- Calculate a vanilla price with strike 100.0

This setup reflects a typical market scenario. We expect the recalibration to be called 2 times: at the beginning and at the end after all volatilities are updated. In particular, we do not want a recalibration at the beginning when the last 3 vanillas are calculated, or in between when volatilities change. The code follows.

```
#include "LazyObject2.h"

void testingLazyObject1(){

        boost::shared_ptr<SimpleVolQuote> v1(new SimpleVolQuote(0.20, 90.0));
        boost::shared_ptr<SimpleVolQuote> v2(new SimpleVolQuote(0.194,95.0));
        boost::shared_ptr<SimpleVolQuote> v3(new SimpleVolQuote(0.187,100.0));
        boost::shared_ptr<SimpleVolQuote> v4(new SimpleVolQuote(0.191,105.0));

        std::vector<boost::shared_ptr<SimpleVolQuote>> volVec;

        volVec.push_back(v1); volVec.push_back(v2);
        volVec.push_back(v3); volVec.push_back(v4);

        boost::shared_ptr<SimpleSmile> mySmile(new SimpleSmile(volVec));

        Time tau=0.5; Real spot=100.0, rd=0.03, rf=0.024;
        Real fwd=spot*std::exp((rd-rf)*tau);

        SabrModel myModel(mySmile, fwd, tau, rd, rf);

        Real res=myModel.getVanillaPrice(100.0);
        std::cout << "Initial Sabr ATM Vanilla Price:" << res << std::endl;
        res=myModel.getVanillaPrice(90.0);
        res=myModel.getVanillaPrice(95.0);
        res=myModel.getVanillaPrice(105.0);

        v1->setVolatility(0.22);
        v2->setVolatility(0.214);
        v3->setVolatility(0.207);
        v4->setVolatility(0.211);

        res=myModel.getVanillaPrice(100.0);
        std::cout << "Last Sabr ATM Vanilla Price:" << res << std::endl;
}
```

The output of this function is

```
Recalibration Performed!
Initial Sabr ATM Vanilla Price:5.39896
Recalibration Performed!
Last Sabr ATM Vanilla Price:5.9546
```

which is exactly the behavior that we wanted.

The following section discusses QuantLib's `Handle` concept. Since this is one of the most important concepts which is used intensely in the library, we will spend some time on its discussion.

In the previous example we have discussed the effects of a parallel smile shift on dependent classes of type `Observer`. To shift the smile we had to invoke the `setVolatility(...)` function on each of the single volatility quotes which were observed by the `SimpleSmile` class, which in turn updated all of its observers. This can become tedious if say 20 volatilities change, and you have to invoke `setVolatility(...)` for each of them. Also, it might make sense to redesign the `SimpleSmile` class to store the volatilities and vectors as simple `std::vector<Real>` objects. The updating procedure would incorporate a reconstruction of a new smile with new volatility/strike vectors. This would require some updating mechanism for the observers since it would not be helpful to reconstruct all classes that depend on `SimpleSmile`, for example 100 different pricers. The alternative would be a `setSmile(...)` function in each of the pricers, but this is not very convenient. Also, it is not very safe as one might forget to set the smile for some pricer which would then return a non market consistent price. Some automatic updating mechanism is clearly required. The consequence is to think about an appropriate concept for such a case.

To achieve this, QuantLib provides a `Handle<T>` class template, where `T` is an `Observable`. Incorporating the concept requires at least 2 functionalities: relinking and updating. The relinking functionality is implemented in the class `RelinkableHandle<T>`, which derives from `Handle<T>`. Relinking replaces the old referenced object with a new one (e.g a new smile object). The updating is incorporated in `Handle<T>` (and consequently in `RelinkableHandle<T>`). It notifies all observers that there is a new object. To summarize:

- a `RelinkableHandle<T>` class is provided, which derives from `Handle<T>`. This class has a public `linkTo(const boost::shared_ptr<T>& h)` function which implements the relinking to a new observable.
- if `T` is updated or relinked, `Handle<T>` propagates the `notifyObservers()` to its own observers. `Handle<T>` can be registered as an Observable.

You might ask, why `RelinkableHandle<T>` is needed and if the same functionality couldn't have been employed by the base class `Handle<T>`. Assume for the moment this would possible and we would have a `DiscountingBondEngine(Handle<YieldTermStructure> curve)` constructor. Assume that the `curve` object is passed to several other classes too. If `Handle<T>` would provide the `linkTo(...)` functionality, the `DiscountingBondEngine` would be able to relink the yield curve such that the object `curve` is now some different yield curve. This would affect all other classes which hold a copy of `curve`. Obviously, these classes would be able to change the curve too, resulting in a non-manageable architecture.

Browsing through the code will show you that many constructors are of type

```
DiscountingBondEngine(Handle<YieldTermStructure> curve,...)
```

to give an example. Deriving `RelinkableHandle<T>` from `Handle<T>` allows to create a relinkable object, for example

```
boost::shared_ptr<YieldTermStructure> ptrCurve(new YieldTermStructure(...));
RelinkableHandle<YieldTermStructure> myHandle(ptrCurve);
```

and construct the original object with

```
DiscountingBondEngine myEngine(myHandle,...);
```

This is valid, since `RelinkableHandle` derives from `Handle`. However, the constructed object `myEngine` cannot relink the object within the class. This relinking is done with

```
boost::shared_ptr<YieldTermStructure> ptrCurveNew(new YieldTermStructure(...));
myHandle.linkTo(ptrCurveNew);
```

The `DiscountingBondEngine` instance `myEngine` will then refer to the new yield term structure, provided it has been registered with `myHandle` before.

Before proceeding, we will rewrite the previously discusses `SimpleDiscountFactor` class with the `Handle` architecture. The class is called `SimpleDiscountFactor2`, the code is shown below

```
class SimpleDiscountFactor2: public Observable, Observer{
private:
        DiscountFactor df_;
        Time mat_;
        Handle<SimpleYield> y_;
public:
        SimpleDiscountFactor2(const Handle<SimpleYield>& y,
                              Time& mat):y_(y),mat_(mat){
                              // register yield as an observable!
                              registerWith(y_);
                              df_=exp(-(*y_)->getYield()*mat_);
        }
        void update(){
                // something has changed, recalculate yield
                df_=exp(-(*y_)->getYield()*mat_);
                notifyObservers();
        }
        Real getDiscount() const{
                return df_;
        }
};
```

This is almost the same architecture as before, but this time, we pass a `Handle<SimpleYield>` to the constructor.

To test the functionality, we will set up a similar scenario as for the original discount factor class. We will change the yield by calling the `setYield()` function first, to test the first update mechanism. We will then relink the original `Handle` to a new one and ask again for the discount factor. This tests the second update mechanism. The function is shown below

```
#include <boost/assign.hpp>
using namespace boost::assign; // bring 'operator+=()' into scope

void testHandle1(){

        boost::shared_ptr<SimpleYield> yield(new SimpleYield(0.04));
        RelinkableHandle<SimpleYield> yieldHandle(yield);
        Time mat=10.0;
        SimpleDiscountFactor2 myDf(yieldHandle,mat);

        std::cout << "Initial Discount:" << myDf.getDiscount() << std::endl;
        yield->setYield(0.06);
        std::cout << "Discount after yield update:" << myDf.getDiscount() << std::endl;
        boost::shared_ptr<SimpleYield> yieldNew(new SimpleYield(0.01));
        yieldHandle.linkTo(yieldNew);
        std::cout << "Discount after relinking:" << myDf.getDiscount() << std::endl;
}
```

The output of the function is

```
Initial Discount:0.67032
Discount after yield update:0.548812
Discount after relinking:0.904837
```

In the implementation of `SimpleDiscountFactor2`, it is shown how to dereference a `Handle` object. At the end we are interested in calling the `getYield()` function. Looking at the `Handle` concept more closly shows that the class represents a pointer to a pointer. The pointer pointing to the observed class can be retrieved with

- `const boost::shared_ptr<T>& currentLink() const;`
- `const boost::shared_ptr<T>& operator->() const;`
- `const boost::shared_ptr<T>& operator*() const;`

This dereference procedures allow us to dereference a handle with `*myHandle` or `myHandle->`, which returns a pointer to the observed object. This pointer has to be dereferenced again to access the basic functions of the observed object.

Thank you!