1. Introduction.

This example shows how to bootstrap a term structure on a number of market rates of different kinds and how to price a simple swap.

Incidentally, it also shows how to use the QuantLib library in one's own code.

2. Here is the general layout of the file `swapvaluation.cpp` which will be generated by this `CWEB` file. The ⟨ include QuantLib header 3 ⟩ and ⟨ **using** declarations for QuantLib classes 5 ⟩ sections are of particular interest since they will be present in all programs using QuantLib.

```
⟨ include QuantLib header 3 ⟩
⟨ include other headers 4 ⟩
⟨ using declarations for QuantLib classes 5 ⟩
⟨ typedefs 19 ⟩
⟨ global variables 15 ⟩
⟨ function declarations 6 ⟩
⟨ function definitions 7 ⟩
⟨ main program 14 ⟩
```

3. In order to use QuantLib, the `ql/quantlib.hpp` header must be included. No other QuantLib headers are necessary as all of them are made available through the latter.

```
⟨ include QuantLib header 3 ⟩ ≡
#include <ql/quantlib.hpp>
```

This code is cited in section 2.

This code is used in section 2.

4. The only other header needed for this example is for output.

```
⟨ include other headers 4 ⟩ ≡
#include <iostream>
```

This code is used in section 2.

5. **using** declarations can be used for easier access to QuantLib classes. A global declaration can be used such as the one below, or individual class names can be imported.

```
⟨ using declarations for QuantLib classes 5 ⟩ ≡
  using namespace QuantLib;
  using QuantLib::Instruments::SimpleSwap;
```

See also sections 16, 27, 30, 32, 34, 36, 48, and 52.

This code is cited in section 2.

This code is used in section 2.

6. For the purpose of this example, we define a few helper functions whose definitions will be given later.

```
⟨ function declarations 6 ⟩ ≡
  void report(const SimpleSwap &);
  Quote make_quote(double value);
  Helper make_deposit_helper(const Quote &, int length, TimeUnit units);
  Helper make_fra_helper(const Quote &, int start, int end);
  Helper make_futures_helper(const Quote &, const Date &imm);
  Helper make_swap_helper(const Quote &, int years);
```

This code is used in section 2.

7. ⟨ function definitions 7 ⟩ ≡
⟨ define *report* 8 ⟩
⟨ define *make_quote* 20 ⟩
⟨ define *make_deposit_helper* 29 ⟩
⟨ define *make_fra_helper* 31 ⟩
⟨ define *make_futures_helper* 33 ⟩
⟨ define *make_swap_helper* 35 ⟩
This code is used in section 2.

8. The function *report* prints swap results to standard output.

⟨ define *report* 8 ⟩ ≡
```
void report(const SimpleSwap &swap) {
  ⟨ print description 9 ⟩
  ⟨ print fixed rate 10 ⟩
  ⟨ print NPV 11 ⟩
  ⟨ print fair rate 12 ⟩
  ⟨ print fair spread 13 ⟩
}
```
This code is used in section 7.

9. The description of the swap was given upon construction and can be retrieved by means of the **Instrument** :: *description* method.

⟨ print description 9 ⟩ ≡
```
std::cout << swap.description() << ":" << std::endl;
```
This code is used in section 8.

10. Rates can be formatted by means of the **RateFormatter** class.

⟨ print fixed rate 10 ⟩ ≡
```
std::cout << "\tFixed␣rate:␣␣" << RateFormatter::toString(swap.fixedRate(),2) << std::endl;
```
This code is used in section 8.

11. The same applies to amounts when the **DoubleFormatter** class is used.

⟨ print NPV 11 ⟩ ≡
```
std::cout << "\tNPV:␣␣␣␣␣␣␣␣␣" << DoubleFormatter::toString(swap.NPV(),2) << std::endl;
```
This code is used in section 8.

12. ⟨ print fair rate 12 ⟩ ≡
```
std::cout << "\tFair␣rate:␣␣␣" << RateFormatter::toString(swap.fairRate(),2) << std::endl;
```
This code is used in section 8.

13. ⟨ print fair spread 13 ⟩ ≡
```
std::cout << "\tFair␣spread:␣" << RateFormatter::toString(swap.fairSpread(),2) << std::endl;
```
This code is used in section 8.

14. Finally, the main program is a sequence of separate tasks which will be the object of the following sections.

```
⟨ main program 14 ⟩ ≡
  int main(int argc, char *argv[])
  {
    try {
      ⟨ collect market data 17 ⟩
      ⟨ bootstrap term structures 37 ⟩
      ⟨ define and price swaps 50 ⟩
      ⟨ perturb the term structures 56 ⟩
      ⟨ reprice the swaps 57 ⟩
      return 0;
    }
    catch(std::exception &e) {
      std::cout << e.what() << std::endl;
      return 1;
    }
    catch(...) {
      std::cout << "unknown error" << std::endl;
      return 1;
    }
  }
```

This code is used in section 2.

15. Setting up. A number of common parameters is used throughout this examples, namely, today's date, the calendar used for date calculations, a common number of settlement days which all instruments in this example will share, and the used currency. For the purpose of this example, we store them as global variables for ease of access.

```
⟨ global variables 15 ⟩ ≡
  Date todaysDate(6, November, 2001);
  Calendar calendar = TARGET();
  int settlementDays = 2;
  Currency currency = EUR;
```

This code is used in section 2.

16. ⟨ **using** declarations for QuantLib classes 5 ⟩ +≡

```
  using QuantLib::Calendars::TARGET;
```

17. The market data we are interested in are the current fixings of a number of deposit, FRA, and swap rates as well as current future prices.

Such rates could be stored in a vector for making it easier to iterate over all the fixings. In this example, they will be stored in separate variables for illustration purposes so that it will be possible to easily refer to any of them by name.

```
⟨ collect market data 17 ⟩ ≡
  ⟨ collect deposit rates 21 ⟩
  ⟨ collect FRA rates 22 ⟩
  ⟨ collect futures prices 23 ⟩
  ⟨ collect swap rates 24 ⟩
```

This code is used in section 14.

18. The **MarketElement** class is used to represent a quote which can vary over time. As we will see, it is able to notify changes so that instruments which depend on its value can recalculate their values.

Also, **MarketElement** is an abstract base class which can be implemented in different ways. Its common interface makes it possible to switch seamlessly between different implementations (databases and live data feeds are among the possibilities which we can imagine). Here, a basic implementation is used called **SimpleMarketElement** whose value can be changed manually. The following function and **typedef** are provided to make it easier to build the desired structure.

19. ⟨ **typedef**s 19 ⟩ ≡

```
  typedef Handle⟨MarketElement⟩ Quote;
```

See also section 28.

This code is used in section 2.

20. ⟨ define *make_quote* 20 ⟩ ≡

```
  Quote make_quote(double value) {
    return Handle⟨MarketElement⟩(new SimpleMarketElement(value));
  }
```

This code is used in section 7.

21. ⟨ collect deposit rates 21 ⟩ ≡
Quote *d1w* = *make_quote*(0.0382);
Quote *d1m* = *make_quote*(0.0372);
Quote *d3m* = *make_quote*(0.0363);
Quote *d6m* = *make_quote*(0.0353);
Quote *d9m* = *make_quote*(0.0348);
Quote *d1y* = *make_quote*(0.0345);
This code is used in section 17.

22. ⟨ collect FRA rates 22 ⟩ ≡
Quote *fra3x6* = *make_quote*(0.037125);
Quote *fra6x9* = *make_quote*(0.037125);
Quote *fra6x12* = *make_quote*(0.037125);
This code is used in section 17.

23. ⟨ collect futures prices 23 ⟩ ≡
Quote *fut1* = *make_quote*(96.2875);
Quote *fut2* = *make_quote*(96.7875);
Quote *fut3* = *make_quote*(96.9875);
Quote *fut4* = *make_quote*(96.6875);
Quote *fut5* = *make_quote*(96.4875);
Quote *fut6* = *make_quote*(96.3875);
Quote *fut7* = *make_quote*(96.2875);
Quote *fut8* = *make_quote*(96.0875);
This code is used in section 17.

24. ⟨ collect swap rates 24 ⟩ ≡
Quote *s2y* = *make_quote*(0.037125);
Quote *s3y* = *make_quote*(0.0398);
Quote *s5y* = *make_quote*(0.0443);
Quote *s10y* = *make_quote*(0.05165);
Quote *s15y* = *make_quote*(0.055175);
This code is used in section 17.

25. Term structure bootstrapping. Term structures are bootstrapped on a number of given rates so that the latter are reproduced when the corresponding instruments are priced on the former.

In particular, the **PiecewiseFlatForward** class is built adding constant instantaneous forward rate intervals corresponding to the underlying instrument maturities. Namely, the first forward rate level is chosen so that the instrument with the shortest maturity is correctly repriced. Then, the instrument with the shortest maturity is chosen among the remaining and the second forward rate level is added. The process is repeated until all instruments have been used.

In order for the process to be extensible, the repricing of the underlying instruments was not built into the term structure itself. Instead, a base class **RateHelper** was created whose derived classes encapsulate rate calculations for different instruments. A number of rate helpers is passed to the term structure to be bootstrapped, which modifies itself until all rate helpers return the desired results.

26. Rate helpers are available for all rates and prices enumerated in the previous sections. For the purpose of this example, a **typedef** and a number of simple functions were defined which make it easier to build the corresponding structures.

27. ⟨ **using** declarations for QuantLib classes 5 ⟩ +≡
```
using QuantLib::TermStructures::RateHelper;
```

28. ⟨ **typedef**s 19 ⟩ +≡
```
typedef Handle⟨RateHelper⟩ Helper;
```

29. Deposit rate helpers take a number of parameters which are described in the QuantLib documentation. We will hard-wire in this function most of them, namely, the one which do not change between different deposits, passing as arguments only the ones which change.

⟨ define *make_deposit_helper* 29 ⟩ ≡
```
Helper make_deposit_helper(const Quote &q, int length, TimeUnit units) {
  static DayCounter counter = Actual360();
  static RollingConvention convention = ModifiedFollowing;
  return Handle⟨RateHelper⟩(new DepositRateHelper(RelinkableHandle⟨MarketElement⟩(q),
      settlementDays, length, units, calendar, convention, counter));
}
```
This code is used in section 7.

30. ⟨ **using** declarations for QuantLib classes 5 ⟩ +≡
```
using QuantLib::DayCounters::Actual360;
using QuantLib::TermStructures::DepositRateHelper;
```

31. The same applies to FRA, futures, and swap rate helpers.

⟨ define *make_fra_helper* 31 ⟩ ≡
```
Helper make_fra_helper(const Quote &q, int start, int end) {
  static DayCounter counter = Actual360();
  static RollingConvention convention = ModifiedFollowing;
  return Handle⟨RateHelper⟩(new FraRateHelper(RelinkableHandle⟨MarketElement⟩(q),
      settlementDays, start, end, calendar, convention, counter));
}
```
This code is used in section 7.

32. ⟨ **using** declarations for QuantLib classes 5 ⟩ +≡
```
using QuantLib::TermStructures::FraRateHelper;
```

33. ⟨ define *make_futures_helper* 33 ⟩ ≡

```
Helper make_futures_helper(const Quote &q, const Date &immDate) {
  static DayCounter counter = Actual360( );
  static RollingConvention convention = ModifiedFollowing;
  static int length = 3;
  return Handle⟨RateHelper⟩(new FuturesRateHelper(RelinkableHandle⟨MarketElement⟩(q),
      immDate, settlementDays, length, calendar, convention, counter));
}
```

This code is used in section 7.

34. ⟨ **using** declarations for QuantLib classes 5 ⟩ +≡

```
using QuantLib::TermStructures::FuturesRateHelper;
```

35. ⟨ define *make_swap_helper* 35 ⟩ ≡

```
Helper make_swap_helper(const Quote &q, int years) {
  static RollingConvention convention = ModifiedFollowing;
  static int fixedLegFrequency = 1;
  static DayCounter counter = Thirty360(Thirty360::European);
  static bool adjusted = false;
  static int floatingLegFrequency = 2;
  return Handle⟨RateHelper⟩(new SwapRateHelper(RelinkableHandle⟨MarketElement⟩(q),
      settlementDays, years, calendar, convention, fixedLegFrequency, adjusted, counter,
      floatingLegFrequency));
}
```

This code is used in section 7.

36. ⟨ **using** declarations for QuantLib classes 5 ⟩ +≡

```
using QuantLib::DayCounters::Thirty360;
using QuantLib::TermStructures::SwapRateHelper;
```

37. Once we have the above machinery, we can easily create term structures by performing the actions outlined below.

⟨ bootstrap term structures 37 ⟩ ≡

```
⟨ create rate helpers 38 ⟩
⟨ combine rate helpers 43 ⟩
⟨ instantiate term structures 47 ⟩
⟨ create common point of access to chosen term structure 49 ⟩
```

This code is used in section 14.

38. Rate helpers are created by using the appropriate helper functions.

⟨ create rate helpers 38 ⟩ ≡

```
⟨ create deposit rate helpers 39 ⟩
⟨ create FRA rate helpers 40 ⟩
⟨ create futures rate helpers 41 ⟩
⟨ create swap rate helpers 42 ⟩
```

This code is used in section 37.

39. ⟨ create deposit rate helpers 39 ⟩ ≡
Helper *d1w_h* = *make_deposit_helper*(*d1w*, 1, *Weeks*);
Helper *d1m_h* = *make_deposit_helper*(*d1m*, 1, *Months*);
Helper *d3m_h* = *make_deposit_helper*(*d3m*, 3, *Months*);
Helper *d6m_h* = *make_deposit_helper*(*d6m*, 6, *Months*);
Helper *d9m_h* = *make_deposit_helper*(*d9m*, 9, *Months*);
Helper *d1y_h* = *make_deposit_helper*(*d1y*, 1, *Years*);
This code is used in section 38.

40. ⟨ create FRA rate helpers 40 ⟩ ≡
Helper *fra3x6_h* = *make_fra_helper*(*fra3x6*, 3, 6);
Helper *fra6x9_h* = *make_fra_helper*(*fra6x9*, 6, 9);
Helper *fra6x12_h* = *make_fra_helper*(*fra6x12*, 6, 12);
This code is used in section 38.

41. ⟨ create futures rate helpers 41 ⟩ ≡
Helper *fut1_h* = *make_futures_helper*(*fut1*, **Date**(19, *December*, 2001));
Helper *fut2_h* = *make_futures_helper*(*fut2*, **Date**(20, *March*, 2002));
Helper *fut3_h* = *make_futures_helper*(*fut3*, **Date**(19, *June*, 2002));
Helper *fut4_h* = *make_futures_helper*(*fut4*, **Date**(18, *September*, 2002));
Helper *fut5_h* = *make_futures_helper*(*fut5*, **Date**(18, *December*, 2002));
Helper *fut6_h* = *make_futures_helper*(*fut6*, **Date**(19, *March*, 2003));
Helper *fut7_h* = *make_futures_helper*(*fut7*, **Date**(18, *June*, 2003));
Helper *fut8_h* = *make_futures_helper*(*fut8*, **Date**(17, *September*, 2003));
This code is used in section 38.

42. ⟨ create swap rate helpers 42 ⟩ ≡
Helper *s2y_h* = *make_swap_helper*(*s2y*, 2);
Helper *s3y_h* = *make_swap_helper*(*s3y*, 3);
Helper *s5y_h* = *make_swap_helper*(*s5y*, 5);
Helper *s10y_h* = *make_swap_helper*(*s10y*, 10);
Helper *s15y_h* = *make_swap_helper*(*s15y*, 15);
This code is used in section 38.

43. Once the rate helpers are created, they can be combined in different ways which will result in different term structures being instantiated. It is not necessary to sort the instrument by maturity before passing them to the term structure constructor; however, one must make sure that no two instruments have the same maturity.

Here, we will create three different combinations.

⟨ combine rate helpers 43 ⟩ ≡
⟨ combine deposits and swaps 44 ⟩
⟨ combine deposits, FRA and swaps 45 ⟩
⟨ combine deposits, futures and swaps 46 ⟩
This code is used in section 37.

44. ⟨ combine deposits and swaps 44 ⟩ ≡
std::**vector**⟨**Helper**⟩ *depoSwaps*;

```
depoSwaps.push_back(d1w_h);
depoSwaps.push_back(d1m_h);
depoSwaps.push_back(d3m_h);
depoSwaps.push_back(d6m_h);
depoSwaps.push_back(d9m_h);
depoSwaps.push_back(d1y_h);
depoSwaps.push_back(s2y_h);
depoSwaps.push_back(s3y_h);
depoSwaps.push_back(s5y_h);
depoSwaps.push_back(s10y_h);
depoSwaps.push_back(s15y_h);
```

This code is used in section 43.

45. ⟨ combine deposits, FRA and swaps 45 ⟩ ≡
std::**vector**⟨**Helper**⟩ *depoFRASwaps*;

```
depoFRASwaps.push_back(d1w_h);
depoFRASwaps.push_back(d1m_h);
depoFRASwaps.push_back(d3m_h);
depoFRASwaps.push_back(fra3x6_h);
depoFRASwaps.push_back(fra6x9_h);
depoFRASwaps.push_back(fra6x12_h);
depoFRASwaps.push_back(s2y_h);
depoFRASwaps.push_back(s3y_h);
depoFRASwaps.push_back(s5y_h);
depoFRASwaps.push_back(s10y_h);
depoFRASwaps.push_back(s15y_h);
```

This code is used in section 43.

46. ⟨ combine deposits, futures and swaps 46 ⟩ ≡
std::**vector**⟨**Helper**⟩ *depoFutSwaps*;

```
depoFutSwaps.push_back(d1w_h);
depoFutSwaps.push_back(d1m_h);
depoFutSwaps.push_back(fut1_h);
depoFutSwaps.push_back(fut2_h);
depoFutSwaps.push_back(fut3_h);
depoFutSwaps.push_back(fut4_h);
depoFutSwaps.push_back(fut5_h);
depoFutSwaps.push_back(fut6_h);
depoFutSwaps.push_back(fut7_h);
depoFutSwaps.push_back(fut8_h);
depoFutSwaps.push_back(s3y_h);
depoFutSwaps.push_back(s5y_h);
depoFutSwaps.push_back(s10y_h);
depoFutSwaps.push_back(s15y_h);
```

This code is used in section 43.

47. Now, term structures can be instantiated by passing the chosen instrument collection as well as a few other parameters. Also, we wrap them into **Handle**s so that they can be safely passed to instruments for pricing.

⟨ instantiate term structures 47 ⟩ ≡
```
DayCounter counter = Actual360();
Handle⟨TermStructure⟩ depoSwapTS(new PiecewiseFlatForward(currency, counter, todaysDate,
    calendar, settlementDays, depoSwaps));
Handle⟨TermStructure⟩ depoFRASwapTS(new PiecewiseFlatForward(currency, counter,
    todaysDate, calendar, settlementDays, depoFRASwaps));
Handle⟨TermStructure⟩ depoFutSwapTS(new PiecewiseFlatForward(currency, counter,
    todaysDate, calendar, settlementDays, depoFutSwaps));
```
This code is used in section 37.

48. ⟨ **using** declarations for QuantLib classes 5 ⟩ +≡
```
using QuantLib::TermStructures::PiecewiseFlatForward;
```

49. Finally, when it is one's intention to switch among a number of term structures, it is necessary to create a **RelinkableHandle** which will act as global point of access for the chosen term structure. Relinking the handle to a different term structure will propagate the change to all its copies as detailed in the QuantLib documentation.

For purpose of illustration, we will create two **RelinkableHandle**s which will give a global link to the term structure used for Euribor fixing and the one used for cash flow discounting, respectively.

⟨ create common point of access to chosen term structure 49 ⟩ ≡
```
RelinkableHandle⟨TermStructure⟩ forecastTS, discountTS;
```
This code is used in section 37.

50. Swap pricing. Once we have a term structure, pricing a simple swap is as easy as instantiating it and asking for its NPV. All that is needed is to instantiate the necessary parameters first.

⟨ define and price swaps 50 ⟩ ≡
⟨ define swap parameters 51 ⟩
⟨ instantiate swaps 53 ⟩
⟨ cycle term structures and output results 54 ⟩
This code is used in section 14.

51. ⟨ define swap parameters 51 ⟩ ≡
double *nominal* = 1000000;
int *length* = 5;
Rate *fixedRate* = 0.04;
bool *payFixedRate* = *true*;
Date *spot* = *calendar*.*advance*(*todaysDate*, *settlementDays*, *Days*);
Date *oneYearHence* = *calendar*.*advance*(*spot*, 1, *Years*);
int *fixedLegFrequency* = 1;
bool *adjusted* = *false*;
RollingConvention *convention* = *ModifiedFollowing*;
DayCounter *swapCounter* = **Thirty360**(**Thirty360**::*European*);
int *floatingLegFrequency* = 2;
Handle⟨**Xibor**⟩ *index*(**new** **Euribor**(6, *Months*, *forecastTS*));
int *fixingDays* = 2;
Spread *spread* = 0.0;
This code is used in section 50.

52. ⟨ **using** declarations for QuantLib classes 5 ⟩ +≡
using **Indexes**::**Xibor**;
using **Indexes**::**Euribor**;

53. ⟨ instantiate swaps 53 ⟩ ≡
SimpleSwap *spot5YearSwap*(*payFixedRate*, *spot*, *length*, *Years*, *calendar*, *convention*, *nominal*, *fixedLegFrequency*, *fixedRate*, *adjusted*, *swapCounter*, *floatingLegFrequency*, *index*, *fixingDays*, *spread*, *discountTS*, "", "5-year␣swap,␣spot");
SimpleSwap *oneYearForward5YearSwap*(*payFixedRate*, *oneYearHence*, *length*, *Years*, *calendar*, *convention*, *nominal*, *fixedLegFrequency*, *fixedRate*, *adjusted*, *swapCounter*, *floatingLegFrequency*, *index*, *fixingDays*, *spread*, *discountTS*, "", "5-year␣swap,␣1␣year␣forward");
This code is used in section 50.

54. Now the swaps can be priced on different term structures just by relinking *forecastTS* and *discountTS* to the desired instances.

⟨ cycle term structures and output results 54 ⟩ ≡
```
std::cout << "*** using depo-swap term structure:" << std::endl;
forecastTS.linkTo(depoSwapTS);
discountTS.linkTo(depoSwapTS);
⟨ output results 55 ⟩
std::cout << "*** using depo-FRA-swap term structure:" << std::endl;
forecastTS.linkTo(depoFRASwapTS);
discountTS.linkTo(depoFRASwapTS);
⟨ output results 55 ⟩
std::cout << "*** using depo-futures-swap term structure:" << std::endl;
forecastTS.linkTo(depoFutSwapTS);
discountTS.linkTo(depoFutSwapTS);
⟨ output results 55 ⟩
```
This code is used in sections 50 and 57.

55. The *report* function is used to output the results. Also, we verify that the 5-years spot swap is correctly repriced.

⟨ output results 55 ⟩ ≡
```
report(spot5YearSwap);
QL_REQUIRE(QL_FABS(spot5YearSwap.fairRate()-s5y->value()) < 1·10^-8, "5 years swap mispriced!");
report(oneYearForward5YearSwap);
```
This code is used in section 54.

56. **MarketElement**s have the ability of notifying changes in their value to their observers. Therefore, we do not need to take any explicit action for the term structures to be bootstrapped again and the swaps to be repriced. To demonstrate this, we simply change the value of the **MarketElement** corresponding to the 5-years swap rate. The explicit downcasting from **Quote** to **Handle**⟨**SimpleMarketElement**⟩ is necessary in order to access the interface of the latter.

⟨ perturb the term structures 56 ⟩ ≡
```
Handle<SimpleMarketElement> fiveYearsRate = s5y;
fiveYearsRate->setValue(0.0460);
std::cout << "*** 5Y swap rate increased to 4.60%" << std::endl;
```
This code is used in section 14.

57. We now simply output again the results. As anticipated, the swaps will now report a different value.

⟨ reprice the swaps 57 ⟩ ≡
```
⟨ cycle term structures and output results 54 ⟩
```
This code is used in section 14.

58. Copyright and license.

59. Index. Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. Error messages are also shown.

The SwapValuation example program