```cpp
#include "Examples/Examples.hpp"

using namespace cfl;

cfl::MultiFunction prb::
cap(const Data::CashFlow & rCap, InterestRateModel & rModel)
{
  //event times: initial time + payment times except the last one
  std::vector<double> uEventTimes(rCap.numberOfPayments);
  uEventTimes.front() = rModel.initialTime();
  std::transform(uEventTimes.begin(), uEventTimes.end()-1,
                 uEventTimes.begin()+1,
                 [&rCap](double dX){ return dX+rCap.period; });
  rModel.assignEventTimes(uEventTimes);

  //last minus one payment time
  int iTime = uEventTimes.size()-1;
  Slice uDiscount =
    rModel.discount(iTime, rModel.eventTimes()[iTime]+rCap.period);
  double dCapFactor = 1.+ rCap.rate*rCap.period;
  //value of the next payment
  Slice uOption = max(1. - uDiscount*dCapFactor, 0.);

  while (iTime > 0) {
    //uOption is the value of future payments
    //we multiply on notional at the end
    iTime--;
    uOption.rollback(iTime);
    uDiscount = rModel.discount(iTime, rModel.eventTimes()[iTime]+rCap.period);
    uOption += max(1. - uDiscount*dCapFactor, 0.);
  }
  uOption *= rCap.notional;

  return interpolate(uOption);
}
```

```cpp
#include "Examples/Examples.hpp"
#include "Examples/ExamplesFunctions.hpp"

using namespace cfl;

cfl::Slice prb::
couponBond(unsigned iTime, const Data::CashFlow & rBond,
           const InterestRateModel & rModel)
{
  Slice uCashFlow = rModel.cash(iTime, 0.);
  double dTime = rModel.eventTimes()[iTime];
  for (unsigned iI=0; iI<rBond.numberOfPayments; iI++) {
    dTime += rBond.period;
    uCashFlow += rModel.discount(iTime, dTime);
  }
  uCashFlow *= (rBond.rate*rBond.period);
  uCashFlow += rModel.discount(iTime, dTime);
  uCashFlow *= rBond.notional;
  return uCashFlow;
}

cfl::Slice prb::
swap(unsigned iTime, const Data::Swap & rSwap,
     const InterestRateModel & rModel)
{
  //assume first that we receive fixed and pay float
  Slice uSwap = couponBond(iTime, rSwap, rModel)-rSwap.notional;
  if (!rSwap.payFloat) { //if we pay fixed
    uSwap *= -1;
  }
  return uSwap;
}


cfl::MultiFunction prb::
swaption(const Data::Swap & rSwap, double dMaturity,
         InterestRateModel & rModel)
{
  PRECONDITION(rModel.initialTime() < dMaturity);

  std::vector<double> uEventTimes(1, rModel.initialTime());
  uEventTimes.push_back(dMaturity);
  rModel.assignEventTimes(uEventTimes);

  int iTime = 1;
  Slice uOption = max(swap(iTime, rSwap, rModel),0);
  uOption.rollback(0);
  return interpolate(uOption);
}
```

```cpp
#include "Examples/Examples.hpp"

using namespace cfl;

cfl::MultiFunction prb::
cancellableCollar(const Data::CashFlow & rCap, double dFloorRate,
                  InterestRateModel & rModel)
{
  std::vector<double> uEventTimes(rCap.numberOfPayments);
  uEventTimes.front() = rModel.initialTime();
  std::transform(uEventTimes.begin(), uEventTimes.end()-1,
                 uEventTimes.begin()+1,
                 [&rCap](double dX){ return dX+rCap.period; });
  rModel.assignEventTimes(uEventTimes);

  //last minus one payment time
  int iTime = uEventTimes.size()-1;
  Slice uDiscount =
    rModel.discount(iTime,rModel.eventTimes()[iTime]+rCap.period);
  double dCapFactor = (1. + rCap.rate*rCap.period);
  double dFloorFactor = (1. + dFloorRate*rCap.period);
  //value of next cap payment
  //we multiply on notional at the end
  Slice uCap = max(1. - uDiscount*dCapFactor, 0.);
  //value of next floor payment
  Slice uFloor = max(uDiscount*dFloorFactor - 1., 0.);
  Slice uOption = (uCap-uFloor);

  while (iTime > 0) {
    //uOption is the value to contiune
    uOption = max(uOption, 0.);
    iTime--;
    uOption.rollback(iTime);
    uDiscount = rModel.discount(iTime,rModel.eventTimes()[iTime]+rCap.period);
    uCap = max(1. - uDiscount*dCapFactor, 0.);
    uFloor = max(uDiscount*dFloorFactor - 1., 0.);
    uOption += (uCap-uFloor);
  }
  uOption *= rCap.notional;

  return interpolate(uOption);
}
```

```cpp
#include "Examples/Examples.hpp"

using namespace cfl;

cfl::MultiFunction prb::
downAndOutCap(const Data::CashFlow & rCap, double dLiborBarrier,
              InterestRateModel & rModel)
{
  std::vector<double> uEventTimes(rCap.numberOfPayments);
  uEventTimes.front() = rModel.initialTime();
  std::transform(uEventTimes.begin(), uEventTimes.end()-1,
                 uEventTimes.begin()+1,
                 [&rCap](double dX){ return dX+rCap.period; });
  rModel.assignEventTimes(uEventTimes);

  //last minus one payment time
  int iTime = uEventTimes.size()-1;
  //express barrier event in terms of discount factors
  double dUpperDiscount = 1./(1. + dLiborBarrier*rCap.period);
  //fixed payment plus notional (as percentage of notional)
  double dFixedFactor = 1. + rCap.period*rCap.rate;
  double dPaymentTime = rModel.eventTimes()[iTime] + rCap.period;
  Slice uDiscount = rModel.discount(iTime, dPaymentTime);
  //value of next payment (we multiply on notional at the end)
  Slice uOption = max(1.-uDiscount*dFixedFactor,0.);

  while (iTime > 0) {
    //uOption is the value to continue (the value of future payments
    //if no barriers have been crossed before and today).
    uOption *= indicator(dUpperDiscount, uDiscount);
    iTime--;
    uOption.rollback(iTime);
    dPaymentTime = rModel.eventTimes()[iTime] + rCap.period;
    uDiscount = rModel.discount(iTime, dPaymentTime);
    //add value of next payment
    uOption += max(1.-uDiscount*dFixedFactor,0.);
  }
  uOption *= rCap.notional;

  return interpolate(uOption);
}
```

```cpp
#include "Examples/Examples.hpp"
#include "Examples/ExamplesFunctions.hpp"

using namespace cfl;

cfl::Slice prb::rate(unsigned iTime, double dPeriod,
                     const cfl::InterestRateModel & rModel)
{
  PRECONDITION(iTime < rModel.eventTimes().size());
  PRECONDITION(dPeriod>cfl::c_dEps);

  double dTime = rModel.eventTimes()[iTime] + dPeriod;
  Slice uDiscount = rModel.discount(iTime, dTime);
  return (1./uDiscount - 1.)/dPeriod;
}

cfl::MultiFunction prb::
futureOnLibor(double dLiborPeriod,
              unsigned iFutureTimes, double dMaturity,
              InterestRateModel & rModel)
{
  PRECONDITION(rModel.initialTime() < dMaturity);
  PRECONDITION(iFutureTimes > 0);

  double dPeriod = (dMaturity - rModel.initialTime())/(iFutureTimes);
  std::vector<double> uEventTimes(iFutureTimes + 1);
  uEventTimes.front() = rModel.initialTime();
  std::transform(uEventTimes.begin(), uEventTimes.end()-1,
                 uEventTimes.begin()+1,
                 [dPeriod](double dX){ return dX+dPeriod; });
  ASSERT(std::abs(uEventTimes.back()- dMaturity) < c_dEps);
  rModel.assignEventTimes(uEventTimes);

  int iTime = rModel.eventTimes().size()-1;
  Slice uFuture = 1.-rate(iTime, dLiborPeriod, rModel);
  while (iTime > 0) {
    //uFuture is the future price today
    iTime--;
    uFuture.rollback(iTime);
    uFuture/=rModel.discount(iTime, rModel.eventTimes()[iTime] + dPeriod);
  }

  return interpolate(uFuture);
}
```