```cpp
#include "SampleExam1/SampleExam1.hpp"

using namespace cfl;

class ForwardStockDividends: public IFunction
{
public:
  ForwardStockDividends(double dSpot, const std::vector<double> & rTimes,
                        const std::vector<double> & rDividends,
                        const Function & rDiscount,
                        double dInitialTime)
    :m_uDiscount(rDiscount), m_uDividends(rDividends), m_uTimes(rTimes)
  {
    PRECONDITION(rTimes.front() > dInitialTime);
    PRECONDITION(rTimes.size() == rDividends.size());
    m_dSpot = dSpot;
    m_dInitialTime = dInitialTime;
  }

  bool belongs(double dT) const
  {
    return (dT >= m_dInitialTime) && (m_uDiscount.belongs(dT))
      && (dT <= m_uTimes.back());
  }

  double operator()(double dTime) const
  {
    PRECONDITION(belongs(dTime));

    unsigned iTime = 0;
    double dPresentValue = m_dSpot;
    while((dTime > m_uTimes[iTime])&&(iTime < m_uTimes.size())) {
      dPresentValue -= (m_uDividends[iTime]) * m_uDiscount(m_uTimes[iTime]);
      iTime++;
    }
    double dForward = dPresentValue / m_uDiscount(dTime);
    return dForward;
  }

private:
  Function m_uDiscount;
  std::vector<double> m_uDividends, m_uTimes;
  double m_dSpot, m_dInitialTime;
};

cfl::Function prb::
forwardStockDividends(double dSpot,
                      std::vector<double> & rFixedDividendsTimes,
                      std::vector<double> & rFixedDividends,
                      const cfl::Function & rDiscount,
                      double dInitialTime)
{
  return Function(new ForwardStockDividends(dSpot, rFixedDividendsTimes,
                                            rFixedDividends, rDiscount, dInitialTime));
}
```

```cpp
#include "SampleExam1/SampleExam1.hpp"

using namespace cfl;

cfl::MultiFunction
prb::upRangeOutPut(double dUpperBarrier, unsigned iOutTimes,
                   const std::vector<double> & rBarrierTimes,
                   double dStrike, double dMaturity,
                   AssetModel & rModel)
{
  PRECONDITION(rModel.initialTime() < rBarrierTimes.front());
  PRECONDITION(rBarrierTimes.back() < dMaturity);
  PRECONDITION(iOutTimes > 0);
  PRECONDITION(iOutTimes <= rBarrierTimes.size());

  std::vector<double> uEventTimes(1, rModel.initialTime());
  uEventTimes.insert(uEventTimes.end(), rBarrierTimes.begin(),
                     rBarrierTimes.end());
  uEventTimes.insert(uEventTimes.end(), dMaturity);
  rModel.assignEventTimes(uEventTimes);

  int iTime = uEventTimes.size()-1;
  Slice uPut = max(dStrike - rModel.spot(iTime), 0.);
  std::vector<Slice> uOption(iOutTimes, uPut);
  iTime--;
  for (unsigned iI=0;iI<uOption.size();iI++) {
    uOption[iI].rollback(iTime);
  }

  while (iTime > 0) {
    //uOption[iI]: the value under the condition that exactly
    //iI barrier events took place before and at iTime.
    Slice uInd = indicator(rModel.spot(iTime),dUpperBarrier);
    for (unsigned iI=0; iI+1<iOutTimes; iI++) {
      uOption[iI] += uInd*(uOption[iI+1]-uOption[iI]);
    }
    uOption.back()*=(1.-uInd);
    iTime--;
    for (unsigned iI=0;iI<uOption.size();iI++) {
      uOption[iI].rollback(iTime);
    }
  }
  return interpolate(uOption.front());
}
```

```cpp
#include "SampleExam1/SampleExam1.hpp"

using namespace cfl;

cfl::Slice
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::MultiFunction prb::
futureOnCheapToDeliver(double dFutureMaturity,
                       unsigned iFutureTimes,
                       const std::vector<cfl::Data::CashFlow> & rBonds,
                       InterestRateModel & rModel)
{
  double dPeriod = (dFutureMaturity - 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; });
  rModel.assignEventTimes(uEventTimes);

  int iTime = rModel.eventTimes().size()-1;
  //select the cheapest to deliver
  Slice uBond = rModel.cash(iTime, std::numeric_limits<double>::max());
  for (unsigned iI = 0; iI<rBonds.size(); iI++) {
    uBond = min(uBond, couponBond(iTime, rBonds[iI], rModel));
  }
  //future price at maturity
  Slice uFuture = uBond;
  while (iTime > 0) {
    //uFuture is the future price today
    iTime--;
    uFuture.rollback(iTime);
    uFuture /= rModel.discount(iTime, rModel.eventTimes()[iTime] + dPeriod);
  }
  return interpolate(uFuture);
}
```

```cpp
#include "SampleExam1/SampleExam1.hpp"

using namespace cfl;

class HistSpot: public IResetValues
{
public:
  HistSpot(const cfl::AssetModel & rModel)
    :m_rModel(rModel)
  {};

  Slice resetValues(unsigned iTime, double dBeforeReset) const
  {
    return m_rModel.spot(iTime);
  }

private:
  const cfl::AssetModel & m_rModel;
};

cfl::PathDependent
histSpot(double dInitialValue, const std::vector<unsigned> & rResetIndexes,
         const AssetModel & rModel)
{
  return PathDependent(new HistSpot(rModel), rResetIndexes,
                       dInitialValue);
}


cfl::MultiFunction prb::
clique(double dMaturity, const std::vector<double> & rAverageTimes,
       const std::vector<double> & rResetTimes, double dInitialStrike,
       AssetModel & rModel)
{
  PRECONDITION(rAverageTimes.front()>rModel.initialTime());
  PRECONDITION(rAverageTimes.back() < dMaturity);

  std::vector<double> uEventTimes(rResetTimes.size()+rAverageTimes.size()+1);
  uEventTimes.front() = rModel.initialTime();
  //add payment times and reset times
  std::vector<double>::iterator itEnd =
    std::set_union(rResetTimes.begin(),rResetTimes.end(),
                   rAverageTimes.begin(),rAverageTimes.end(),
                   uEventTimes.begin()+1);
  uEventTimes.resize(itEnd-uEventTimes.begin());
  rModel.assignEventTimes(uEventTimes);

  std::vector<unsigned> uResetIndexes(rResetTimes.size());
  for (unsigned iI=0; iI<uResetIndexes.size(); iI++) {
    uResetIndexes[iI] =
      std::lower_bound(uEventTimes.begin(), uEventTimes.end(),
                       rResetTimes[iI]) - uEventTimes.begin();
  }
  unsigned iStrike =
    rModel.addState(histSpot(dInitialStrike,uResetIndexes, rModel));

  int iTime = rModel.eventTimes().size()-1;
  Slice uOption = rModel.cash(iTime, 0.);
  while (iTime > 0) {
    //uOption is the present value of all future call payoffs in the sum
    if (std::binary_search(rAverageTimes.begin(), rAverageTimes.end(),
                           rModel.eventTimes()[iTime])) {
      uOption += max(rModel.spot(iTime)-rModel.state(iTime, iStrike),0.)
        *rModel.discount(iTime, dMaturity);
    }
    iTime--;
    uOption.rollback(iTime);
  }
  //not forgetting to divide on number of averaging times
  uOption /= rAverageTimes.size();

  return interpolate(uOption, iStrike);
```

```
}
```