```cpp
#include <numeric>
#include "SampleExam1/SampleExam1.hpp"
#include "cfl/Macros.hpp"
#include <cmath>

using namespace cfl;
using namespace std;

cfl::Function prb::yieldVasicek(double dTheta, double dLambda,
                                double dSigma, double dR0,  double dInitialTime)
{
  PRECONDITION((dLambda >= 0) && (dSigma>=0));

  std::function<double(double)> uY =
    [dTheta, dLambda, dSigma, dR0, dInitialTime](double dT) {
      double dZ = dT-dInitialTime;
      double dX = std::max(dLambda*dZ, cfl::EPS);
      double dB = (1.-std::exp(-dX))/dX;
      double dR = dSigma*dZ/dX;
      double dC = (1. - std::exp(-2.*dX))/(2.*dX);
      double dY = dR0*dB + dTheta*dZ*(1.-dB)/dX + 0.5*dR*dR*(1.-2.*dB + dC);
      return dY;
    };
  return Function(uY, dInitialTime);
}
```

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

using namespace cfl;
using namespace std;

cfl::Function prb::
forwardFXLogLinInterp(double dSpotFX,
                      const std::vector< double > & rTimes,
                      const std::vector< double > & rDom,
                      const std::vector< double > & rFor,
                      double dInitialTime
                      )
{
  PRECONDITION(rTimes.size() == rDom.size());
  PRECONDITION(rTimes.size() == rFor.size());
  PRECONDITION(rTimes.size()>0);
  PRECONDITION(rTimes.front()>dInitialTime);
  PRECONDITION(std::is_sorted(rTimes.begin(), rTimes.end(),
                              std::less_equal<double>()));

  //times for interpolation: initial time + discount maturities
  std::vector<double> uTimes(rTimes.size()+1);
  uTimes.front() = dInitialTime;
  copy(rTimes.begin(), rTimes.end(), uTimes.begin()+1);

  //args for interpolation: logs of forward exchange rates
  std::vector<double> uLogForw(uTimes.size());
  uLogForw.front() = std::log(dSpotFX);
  std::transform(rDom.begin(), rDom.end(), rFor.begin(), uLogForw.begin()+1,
                 [dSpotFX](double dDom, double dFor)
                 {
                   return std::log(dSpotFX*dFor/dDom);
                 });

  //linear interpolation
  cfl::Interp uLinear = NInterp::linear();
  Function uLogForwFunction =
    uLinear.interpolate(uTimes.begin(), uTimes.end(), uLogForw.begin());

  return exp(uLogForwFunction);
}
```

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

using namespace cfl;
using namespace std;

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());
  PRECONDITION(std::is_sorted(rBarrierTimes.begin(), rBarrierTimes.end(),
                              std::less_equal<double>()));

  std::vector<double> uEventTimes(rBarrierTimes.size()+2);
  uEventTimes.front() = rModel.initialTime();
  copy(rBarrierTimes.begin(), rBarrierTimes.end(), uEventTimes.begin()+1);
  uEventTimes.back() = 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;
using namespace std;

namespace NFuturesCheapDeliver
{
  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;
  }
} // namespace NFuturesCheapDeliver

using namespace NFuturesCheapDeliver;

cfl::MultiFunction prb::
    futuresOnCheapToDeliver(double dFuturesMaturity,
                            unsigned iFuturesTimes,
                            const std::vector<cfl::Data::CashFlow> &rBonds,
                            InterestRateModel &rModel)
{
  double dPeriod = (dFuturesMaturity - rModel.initialTime()) / (iFuturesTimes);
  std::vector<double> uEventTimes(iFuturesTimes + 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 uFutures = uBond;
  while (iTime > 0)
  {
    //uFutures is the future price today
    iTime--;
    uFutures.rollback(iTime);
    uFutures /= rModel.discount(iTime, rModel.eventTimes()[iTime] + dPeriod);
  }
  return interpolate(uFutures);
}
```