```cpp
#include "cfl/Interp.hpp"
#include "SampleExam2/SampleExam2.hpp"

using namespace cfl;


cfl::Function
prb::discountSwapLogLinearInterp(const std::vector<double> & rSwapRates,
                                 double dPeriod, double dInitialTime)
{
  PRECONDITION(rSwapRates.size() > 0);

  //times for interpolation = initial time + payment times
  std::vector<double> uTimes(rSwapRates.size()+1);
  uTimes.front() = dInitialTime;
  std::transform(uTimes.begin(), uTimes.end()-1, uTimes.begin()+1,
                 [dPeriod](double dX){ return dX+dPeriod; });

  std::vector<double> uDiscount(uTimes.size());
  uDiscount.front() = 1.;
  double dSum = 0;
  for (unsigned iI=1; iI<uDiscount.size(); iI++) {
    uDiscount[iI] =
      (1 - dSum*rSwapRates[iI-1]*dPeriod)/(1 + rSwapRates[iI-1]*dPeriod);
    dSum += uDiscount[iI];
  }
  std::vector<double> uLogDiscount(uDiscount.size());
  std::transform(uDiscount.begin(), uDiscount.end(), uLogDiscount.begin(),
                 [](double dX){ return std::log(dX); });

  //linear interpolation of the logarithm of discount factors
  cfl::Interp uLinear = NInterp::linear();
  Function uLogDiscountFunction =
    uLinear.interpolate(uTimes.begin(), uTimes.end(), uLogDiscount.begin());

  return cfl::exp(uLogDiscountFunction);
}
```

```cpp
#include "SampleExam2/SampleExam2.hpp"

using namespace cfl;

cfl::MultiFunction
prb::boost(double dNotional, double dLowerBarrier,
           double dUpperBarrier,
           const std::vector<double> & rBarrierTimes,
           AssetModel & rModel)
{
  PRECONDITION(rBarrierTimes.front() > rModel.initialTime());
  PRECONDITION(dLowerBarrier < dUpperBarrier);

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

  unsigned iTime = rModel.eventTimes().size()-1;
  ASSERT(iTime == rBarrierTimes.size());
  Slice uOption = rModel.cash(iTime, dNotional);
  while (iTime > 0) {
    //uOption is the value to continue
    //payoff if stop today
    double dPayoff = dNotional*(iTime-1.)/rBarrierTimes.size();
    Slice uIndStop = indicator(dLowerBarrier, rModel.spot(iTime)) +
      indicator(rModel.spot(iTime), dUpperBarrier);
    uOption += (dPayoff - uOption)*uIndStop;
    iTime--;
    uOption.rollback(iTime);
  }
  return interpolate(uOption);
}
```

```cpp
#include "SampleExam2/SampleExam2.hpp"

using namespace cfl;

cfl::MultiFunction prb::
resetCouponPutBond(const Data::CashFlow & rBond,
                   double dResetCouponRate, double dRedemptionPrice,
                   InterestRateModel & rModel)
{
  ASSERT(dRedemptionPrice < 1);
  ASSERT(dResetCouponRate < rBond.rate);
  ASSERT(rBond.numberOfPayments > 1);

  std::vector<double> uEventTimes(rBond.numberOfPayments);
  uEventTimes.front() = rModel.initialTime();
  std::transform(uEventTimes.begin(), uEventTimes.end()-1,
                 uEventTimes.begin()+1,
                 [&rBond](double dX){ return dX+rBond.period; });
  rModel.assignEventTimes(uEventTimes);

  //last minus one coupon time
  int iTime = uEventTimes.size()-1;
  double dOriginalCoupon =  rBond.notional * rBond.rate * rBond.period;
  double dResetCoupon = rBond.notional * dResetCouponRate * rBond.period;
  double dRedemptionValue = dRedemptionPrice * rBond.notional;
  Slice uDiscount =
    rModel.discount(iTime,rModel.eventTimes()[iTime]+rBond.period);
  Slice uBondBeforeReset = uDiscount*(rBond.notional + dOriginalCoupon);
  Slice uBondAfterReset = uDiscount*(rBond.notional + dResetCoupon);

  while (iTime > 0) {
    uBondAfterReset = max(uBondAfterReset, dRedemptionValue);
    uBondBeforeReset = min(uBondAfterReset, uBondBeforeReset);
    uBondAfterReset+= dResetCoupon;
    uBondBeforeReset+=dOriginalCoupon;
    iTime--;
    uBondBeforeReset.rollback(iTime);
    uBondAfterReset.rollback(iTime);
  }

  return interpolate(uBondBeforeReset);
}
```

```cpp
#include "SampleExam2/SampleExam2.hpp"

using namespace cfl;

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

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

class HistRate: public IResetValues
{
public:
  HistRate(double dPeriod, const InterestRateModel & rModel)
    :m_dPeriod(dPeriod), m_rModel(rModel)
  {}

  Slice resetValues(unsigned iTime, double dBeforeReset) const
  {
    return rate(iTime, m_dPeriod, m_rModel);
  }

private:
  double m_dPeriod;
  const InterestRateModel & m_rModel;
};

cfl::PathDependent
histRate(double dPeriod, double dInitialRate,
         const std::vector<unsigned> & rResetTimes,
         const cfl::InterestRateModel & rModel)
{
  return PathDependent(new HistRate(dPeriod, rModel),
                       rResetTimes, dInitialRate);
}

cfl::MultiFunction prb::
resetCap(const Data::CashFlow & rCap, double dSpread,
         const std::vector<double> & rResetTimes, InterestRateModel & rModel)
{
  PRECONDITION(rResetTimes.front() > rModel.initialTime());

  //cap times include initial time plus all payment times except the last one
  std::vector<double> uCapTimes(rCap.numberOfPayments, rModel.initialTime());
  std::transform(uCapTimes.begin(), uCapTimes.end()-1, uCapTimes.begin()+1,
                 [&rCap](double dX){ return dX+rCap.period; });

  ASSERT(uCapTimes.back() >= rResetTimes.back());

  std::vector<double> uEventTimes(uCapTimes.size()+rResetTimes.size());
  std::vector<double>::iterator itEnd =
    std::set_union(uCapTimes.begin(), uCapTimes.end(),
                   rResetTimes.begin(), rResetTimes.end(),
                   uEventTimes.begin());
  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 iState =
    rModel.addState(histRate(rCap.period, rCap.rate-dSpread, uResetIndexes,
                             rModel));
  //last minus one payment time
  int iTime = uEventTimes.size()-1;
```

```
  double dTime = rModel.eventTimes()[iTime];
  ASSERT(std::binary_search(uCapTimes.begin(),uCapTimes.end(),dTime));
  Slice uDiscount = rModel.discount(iTime, dTime + rCap.period);
  //current value of next float payment
  Slice uFloat = 1. - uDiscount;
  //current value of next fixed payment
  Slice uFixed =
    uDiscount * (rModel.state(iTime, iState) + dSpread)*rCap.period;
  Slice uCap = max(uFloat-uFixed, 0);

  while (iTime > 0) {
    //uCap is the value of future payments
    iTime--;
    uCap.rollback(iTime);
    dTime = rModel.eventTimes()[iTime];
    if (std::binary_search(uCapTimes.begin(), uCapTimes.end(), dTime)) {
      //the current time is a cap time
      uDiscount = rModel.discount(iTime,dTime+rCap.period);
      uFloat = 1.-uDiscount;
      uFixed = uDiscount*(rModel.state(iTime, iState)+dSpread)*rCap.period;
      uCap += max(uFloat-uFixed, 0);
    }
  }
  uCap *= rCap.notional;

  return interpolate(uCap, iState);
}
```