```cpp
#include <numeric>
#include "SampleExam2/SampleExam2.hpp"
#include "cfl/Interp.hpp"
#include "cfl/Macros.hpp"
#include <cmath>

using namespace cfl;
using namespace std;

namespace NVolVarLinInterp
{
  Function volatilityVar(const Function &rVar, double dInitialTime)
  {
    std::function<double(double)> uVol =
        [dInitialTime, rVar](double dT) {
          PRECONDITION(dT >= dInitialTime);
          double dX = std::max<double>(dT - dInitialTime, cfl::EPS);
          return std::sqrt(rVar(dInitialTime + dX) / dX);
        };

    std::function<bool(double)> uVolBelongs =
        [rVar](double dT) {
          return rVar.belongs(dT);
        };

    return Function(uVol, uVolBelongs);
  }
} // namespace NVolVarLinInterp

using namespace NVolVarLinInterp;

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

  std::vector<double> uVar(rVol.size() + 1);
  uVar.front() = 0.;
  std::transform(rTimes.begin(), rTimes.end(), rVol.begin(),
                 uVar.begin() + 1,
                 [dInitialTime](double dT, double dVol) {
                   return dVol * dVol * (dT - dInitialTime);
                 });

  std::vector<double> uTimes(rTimes.size() + 1);
  uTimes.front() = dInitialTime;
  copy(rTimes.begin(), rTimes.end(), uTimes.begin() + 1);

  Interp uInterp = cfl::NInterp::linear();
  Function uVarFunc =
      uInterp.interpolate(uTimes.begin(), uTimes.end(), uVar.begin());

  return volatilityVar(uVarFunc, dInitialTime);
}
```

```cpp
#include <numeric>
#include "SampleExam2/SampleExam2.hpp"

using namespace cfl;
using namespace std;

namespace NDiscountHullWhiteFit
{
  cfl::Function yieldShape1(double dLambda, double dInitialTime)
  {
    std::function<double(double)> uShape =
        [dLambda, dInitialTime](double dT) {
          double dX = std::max(dLambda * (dT - dInitialTime), cfl::EPS);
          return (1 - std::exp(-dX)) / dX;
        };
    return Function(uShape, dInitialTime);
  }

  cfl::Function
      discountYieldFit(const std::vector<double> &rTime, const std::vector<double> &rDF,
                       double dInitialTime, cfl::Fit &rFit, Function &rErr)
  {
    PRECONDITION(rTime.size() == rDF.size());
    PRECONDITION(rTime.size() > 0);
    PRECONDITION(rTime.front() > dInitialTime);
    PRECONDITION(std::is_sorted(rTime.begin(), rTime.end(),
                                std::less_equal<double>()));

    std::vector<double> uYield(rTime.size());
    std::transform(rTime.begin(), rTime.end(), rDF.begin(), uYield.begin(),
                   [dInitialTime](double dT, double dD) {
                     ASSERT(dT - dInitialTime > cfl::EPS);
                     return -std::log(dD) / (dT - dInitialTime);
                   });
    rFit.assign(rTime.begin(), rTime.end(), uYield.begin());

    Function uT([dInitialTime](double dT) { return dT - dInitialTime; },
                dInitialTime);
    Function uDF = exp(-rFit.fit() * uT);

    rErr = rFit.err() * uDF * uT;

    return uDF;
  }
} // namespace NDiscountHullWhiteFit

using namespace NDiscountHullWhiteFit;

cfl::Function prb::
    discountHullWhiteFit(const std::vector<double> &rTimes,
                         const std::vector<double> &rDF,
                         double dLambda, double dInitialTime,
                         cfl::Function &rErr, cfl::FitParam &rParam)
{
  PRECONDITION(dLambda >= 0);
  PRECONDITION(rTimes.front() > dInitialTime);
  PRECONDITION(is_sorted(rTimes.begin(), rTimes.end(), std::less_equal<double>()));

  std::vector<Function> uF = {Function(1.), yieldShape1(dLambda, dInitialTime)};
  Fit uFit = cfl::NFit::linear(uF);
  Function uD = discountYieldFit(rTimes, rDF, dInitialTime, uFit, rErr);
  rParam = uFit.param();
  return uD;
}
```

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

using namespace cfl;
using namespace std;

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

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

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);
}
```