```cpp
#include <numeric>
#include "SampleExam3/SampleExam3.hpp"
#include "cfl/Interp.hpp"

using namespace cfl;
using namespace cfl::Data;

class VolatilityFromVariance: public cfl::IFunction
{
public:
  VolatilityFromVariance(const cfl::Function & rVariance, double dInitialTime)
    :m_uVariance(rVariance), m_dInitialTime(dInitialTime)
  {}

  double operator()(double dT) const
  {
    PRECONDITION(belongs(dT));
    double dVolatility;
    double dTime = std::max(dT, m_dInitialTime + cfl::c_dEps);
    dVolatility = std::sqrt(m_uVariance(dTime)/(dTime - m_dInitialTime));
    return dVolatility;
  }

  bool belongs(double dT) const
  {
    return m_uVariance.belongs(dT);
  }

private:
  cfl::Function m_uVariance;
  double m_dInitialTime;
};

cfl::Function prb::
volatilityLinearInterpOfVar(const std::vector<double> & rTimes,
                            const std::vector<double> & rVolatilities,
                            double dInitialTime)
{
  PRECONDITION(rTimes.size() == rVolatilities.size());
  PRECONDITION(rTimes.size()>0);
  PRECONDITION(rTimes.front()>dInitialTime);

  std::vector<double> uVariance(rVolatilities.size()+1,0.);
  std::transform(rTimes.begin(), rTimes.end(), rVolatilities.begin(),
                 uVariance.begin()+1, [dInitialTime](double dT, double dVol) {
                   return dVol*dVol*(dT-dInitialTime); });
  std::vector<double> uTimes(rTimes);
  uTimes.insert(uTimes.begin(), dInitialTime);
  Interp uLinear = cfl::NInterp::linear();
  Function uVar =
    uLinear.interpolate(uTimes.begin(),uTimes.end(),uVariance.begin());

  return cfl::Function(new VolatilityFromVariance(uVar, dInitialTime));
}
```

```cpp
#include "SampleExam3/SampleExam3.hpp"

using namespace cfl;

cfl::MultiFunction prb::
compoundCallAmericanPut(double dMaturity, double dCompoundCallStrike,
                        double dPutStrike,
                        const std::vector<double> & rExerciseTimes,
                        AssetModel & rModel)
{
  PRECONDITION(rModel.initialTime() < dMaturity);
  PRECONDITION(dMaturity < rExerciseTimes.front());

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

  int iTime = uEventTimes.size()-1;
  Slice uPut = rModel.cash(iTime, 0.);
  while (iTime > 1) {
    uPut = max(uPut, dPutStrike - rModel.spot(iTime));
    iTime--;
    uPut.rollback(iTime);
  }
  ASSERT(uEventTimes[iTime] == dMaturity);
  Slice uCall = max(uPut - dCompoundCallStrike, 0.);
  uCall.rollback(0);
  return interpolate(uCall);
}
```

```cpp
#include "SampleExam3/SampleExam3.hpp"

using namespace cfl;

cfl::MultiFunction prb::
callableCappedFloater(const Data::CashFlow & rCap, double dLiborSpread,
                      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 = rModel.eventTimes().size() - 1;
  Slice uDiscount =
    rModel.discount(iTime, rModel.eventTimes()[iTime] + rCap.period);
  double dSpreadFactor = rCap.notional*(dLiborSpread*rCap.period - 1.);
  double dCapFactor = rCap.notional*rCap.rate*rCap.period;
  Slice uValueOfNextCoupon =
    min(rCap.notional + dSpreadFactor*uDiscount, dCapFactor*uDiscount);
  //uOption is the value to continue (includes next coupon)
  Slice uOption = rCap.notional*uDiscount + uValueOfNextCoupon;
  while (iTime >0) {
    //uOption is the value to continue (includes the value of the next coupon).
    uOption = min(uOption, rCap.notional);
    iTime--;
    uOption.rollback(iTime);
    uDiscount =
      rModel.discount(iTime, rModel.eventTimes()[iTime] + rCap.period);
    uValueOfNextCoupon =
      min(rCap.notional + dSpreadFactor*uDiscount, dCapFactor*uDiscount);
    uOption += uValueOfNextCoupon;
  }
  return interpolate(uOption);
}
```

```cpp
#include "SampleExam3/SampleExam3.hpp"

using namespace cfl;

Slice 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;
}

class AmortizingNotional: public cfl::IResetValues
{
public:
  AmortizingNotional(double (*fAm)(double), double dPeriod,
                     const cfl::InterestRateModel & rModel)
    :m_rModel(rModel), m_fAm(fAm), m_dPeriod(dPeriod)
  {}
  Slice resetValues(unsigned iTime, double dBeforeReset) const {
    Slice uRate = rate(iTime,m_dPeriod,m_rModel);
    Slice uAm  = uRate.apply(m_fAm);
    return dBeforeReset*uAm;
  }
private:
  const cfl::InterestRateModel & m_rModel;
  double (*m_fAm)(double);
  double m_dPeriod;
};

cfl::PathDependent
amortizingNotional(double (*fAm)(double), double dPeriod,
                   const std::vector<unsigned> & rIndexes,
                   const InterestRateModel & rModel)
{
  double dOrigin = 1.;
  return PathDependent(new AmortizingNotional(fAm, dPeriod, rModel),
                       rIndexes, dOrigin);
}

cfl::MultiFunction prb::
indexAmortizingSwap(const Data::Swap & rSwap, double (*fAmortizing)(double),
                    double dLowerThreshold, InterestRateModel & rModel)
{
  std::vector<double>
    uEventTimes(rSwap.numberOfPayments, rModel.initialTime());
  std::transform(uEventTimes.begin(), uEventTimes.end()-1,
                 uEventTimes.begin()+1,
                 [&rSwap](double dX){ return dX+rSwap.period; });
  rModel.assignEventTimes(uEventTimes);

  std::vector<unsigned> uIndexes(uEventTimes.size()-1, 1);
  std::transform(uIndexes.begin(), uIndexes.end()-1, uIndexes.begin()+1,
                 [](unsigned iX){ return iX+1; });
  int iNotional =
    rModel.addState(amortizingNotional(fAmortizing, rSwap.period,
                                       uIndexes,rModel));

  //last payment time before maturity
  int iTime = uEventTimes.size()-1;
  Slice uSwap = rModel.cash(iTime, 0.);
  double dFixedFactor = 1.+rSwap.rate * rSwap.period;
  //percentage of amortizing notional
  Slice uNotional = rModel.state(iTime, iNotional);
  Slice uDiscount =
    rModel.discount(iTime, rModel.eventTimes()[iTime] + rSwap.period);
  //We pay fixed and receive float. We multiply on notional at the end.
  uSwap += uNotional * (1. - uDiscount*dFixedFactor);
```

```
  while (iTime >0) {
    //uSwap is the value of future payments if we continue.
    //We stop if notional is below threshold.
    uSwap *= indicator(uNotional, dLowerThreshold);
    iTime--;
    uSwap.rollback(iTime);
    uNotional = rModel.state(iTime, iNotional);
    uDiscount =
      rModel.discount(iTime, rModel.eventTimes()[iTime] + rSwap.period);
    //We pay fixed. Add the value of next payment.
    uSwap += uNotional * (1. - uDiscount*dFixedFactor);
  }

  if (rSwap.payFloat) {
    uSwap *= -1.;
  }
  uSwap *= rSwap.notional;

  return interpolate(uSwap, iNotional);
}
```