```cpp
#include "Session4/Session4.hpp"

using namespace cfl;
using namespace std;

cfl::MultiFunction prb::
    floor(const Data::CashFlow &rFloor, InterestRateModel &rModel)
{
  std::vector<double> uEventTimes(rFloor.numberOfPayments);
  uEventTimes.front() = rModel.initialTime();
  std::transform(uEventTimes.begin(), uEventTimes.end() - 1,
                 uEventTimes.begin() + 1,
                 [&rFloor](double dX) { return dX + rFloor.period; });
  rModel.assignEventTimes(uEventTimes);

  //last minus one payment
  int iTime = uEventTimes.size() - 1;
  //floor payment plus notional (as percentage)
  double dFloorFactor = 1. + rFloor.rate * rFloor.period;
  double dPaymentTime = uEventTimes[iTime] + rFloor.period;
  Slice uDiscount = rModel.discount(iTime, dPaymentTime);
  //value of next payment (we multiply on notional at the end)
  Slice uOption = max(uDiscount * dFloorFactor - 1., 0.);

  while (iTime > 0)
  {
    //uOption is the value of future payments
    iTime--;
    uOption.rollback(iTime);
    dPaymentTime = uEventTimes[iTime] + rFloor.period;
    uDiscount = rModel.discount(iTime, dPaymentTime);
    uOption += max(uDiscount * dFloorFactor - 1., 0.);
  }
  uOption *= rFloor.notional;

  return interpolate(uOption);
}
```

```cpp
#include "Session4/Session4.hpp"

using namespace cfl;
using namespace std;

cfl::MultiFunction prb::
    putOnFRA(double dFixedRate, double dPeriod, double dNotional,
             double dMaturity, InterestRateModel &rModel)
{
  PRECONDITION(rModel.initialTime() < dMaturity);

  std::vector<double> uEventTimes = {rModel.initialTime(), dMaturity};
  rModel.assignEventTimes(uEventTimes);

  int iTime = uEventTimes.size() - 1;
  //multiply on notional at the end
  double dFixedPayment = 1. + dFixedRate * dPeriod;
  Slice uOption =
      max(1. - rModel.discount(iTime, dMaturity + dPeriod) * dFixedPayment, 0.);
  uOption.rollback(0);
  uOption *= dNotional;

  return interpolate(uOption);
}
```

```cpp
#include "Session4/Session4.hpp"

using namespace cfl;
using namespace std;

namespace NCapOnSwapRate
{
  cfl::Slice
  swapRate(unsigned iTime, double dPeriod,
           unsigned iPeriods, const cfl::InterestRateModel &rModel)
  {
    cfl::Slice uFixed = rModel.cash(iTime, 0.);
    double dTime = rModel.eventTimes()[iTime];
    for (unsigned iI = 0; iI < iPeriods; iI++)
    {
      dTime += dPeriod;
      uFixed += rModel.discount(iTime, dTime);
    }
    uFixed *= dPeriod;
    cfl::Slice uFloat = 1. - rModel.discount(iTime, dTime);
    cfl::Slice uRate = (uFloat / uFixed);
    return uRate;
  }
}

using namespace NCapOnSwapRate;

cfl::MultiFunction prb::
    capOnSwapRate(const Data::CashFlow &rCap,
                  double dSwapPeriod, unsigned iSwapPayments,
                  InterestRateModel &rModel)
{
  //event times: initial time + payment times - maturity
  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 = uEventTimes.size() - 1;
  Slice uDiscount =
      rModel.discount(iTime, rModel.eventTimes()[iTime] + rCap.period);
  Slice uOption = uDiscount * max(swapRate(iTime, dSwapPeriod, iSwapPayments, rModel) - rCap.rate
, 0.);

  while (iTime > 0)
  {
    //uOption is the value of future payments
    //we multiply on rCap.notional*rCap.period at the end
    iTime--;
    uOption.rollback(iTime);
    uDiscount = rModel.discount(iTime, rModel.eventTimes()[iTime] + rCap.period);
    uOption += uDiscount * max(swapRate(iTime, dSwapPeriod, iSwapPayments, rModel) - rCap.rate, 0
.);
  }
  uOption *= (rCap.notional * rCap.period);

  return interpolate(uOption);
}
```

```cpp
#include "Session4/Session4.hpp"

using namespace cfl;
using namespace std;

namespace NCancelSwapArrears
{
  cfl::Slice
  floatInterest(unsigned iTime, double dPeriod, cfl::InterestRateModel &rModel)
  {
    double dMaturity = rModel.eventTimes()[iTime] + dPeriod;
    return (1. / rModel.discount(iTime, dMaturity) - 1.);
  }
}

using namespace NCancelSwapArrears;

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

  int iTime = uEventTimes.size() - 1;
  Slice uOption = rModel.cash(iTime, 0.);
  double dFixedPayment = rSwap.rate * rSwap.period;
  while (iTime > 0)
  {
    //uOption is the value to continue
    //we multiply on notional at the end
    uOption = max(uOption, 0.);
    Slice uCurrentPayment =
        floatInterest(iTime, rSwap.period, rModel) - dFixedPayment;
    if (rSwap.payFloat)
    {
      uCurrentPayment *= -1.;
    }
    uOption += uCurrentPayment;
    iTime--;
    uOption.rollback(iTime);
  }
  uOption *= rSwap.notional;
  return interpolate(uOption);
}
```