```cpp
#include "Homework3/Homework3.hpp"

using namespace cfl;
using namespace std;

cfl::MultiFunction
prb::straddle(double dStrike, double dMaturity,
                              AssetModel & rModel)
{
  PRECONDITION(rModel.initialTime() < dMaturity);

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

  int iEventTime = 1;
  Slice uOption = abs(dStrike - rModel.spot(iEventTime));
  uOption.rollback(0);
  return interpolate(uOption);
}
```

```cpp
#include "Homework3/Homework3.hpp"

using namespace cfl;
using namespace std;

namespace NAmerCallForw
{
  Slice longForward(double dForwardPrice, double dTimeToMaturity,
                    unsigned iTime, const AssetModel &rModel)
  {
    double dMaturity = rModel.eventTimes()[iTime] + dTimeToMaturity;
    Slice uForward = rModel.discount(iTime, dMaturity) * (rModel.forward(iTime, dMaturity) - dFor
wardPrice);
    return uForward;
  }
}

using namespace NAmerCallForw;

cfl::MultiFunction prb::
    americanCallOnForward(double dForwardPrice,
                          double dTimeToMaturity,
                          const std::vector<double> &rExerciseTimes,
                          AssetModel &rModel)
{
  PRECONDITION(rModel.initialTime() < rExerciseTimes.front());
  PRECONDITION(std::is_sorted(rExerciseTimes.begin(), rExerciseTimes.end(),
                              std::less_equal<double>()));

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

  int iTime = uEventTimes.size() - 1;
  Slice uOption = rModel.cash(iTime, 0.);
  while (iTime > 0)
  {
    //uOption is the value to continue
    uOption = max(uOption, longForward(dForwardPrice, dTimeToMaturity,
                                       iTime, rModel));
    iTime--;
    uOption.rollback(iTime);
  }
  return interpolate(uOption);
}
```

```cpp
#include "Homework3/Homework3.hpp"

using namespace cfl;
using namespace std;

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

  int iTime = uEventTimes.size() - 1;
  Slice uOption = rModel.cash(iTime, 0.);
  while (iTime > 0)
  {
    uOption +=
        (dNotional - uOption) * indicator(dLowerBarrier, rModel.spot(iTime));
    iTime--;
    uOption.rollback(iTime);
  }

  return interpolate(uOption);
}
```

```cpp
#include "Homework3/Homework3.hpp"

using namespace cfl;
using namespace std;

cfl::MultiFunction prb::
upInAmericanPut(double dUpperBarrier,
                    const std::vector<double> & rBarrierTimes,
                    double dStrike, const std::vector<double> & rExerciseTimes,
                    AssetModel & rModel)
{
  PRECONDITION(rModel.initialTime() < rBarrierTimes.front());
  PRECONDITION(rBarrierTimes.front() < rExerciseTimes.front());
  PRECONDITION(rBarrierTimes.back() < rExerciseTimes.back());
  PRECONDITION(std::is_sorted(rExerciseTimes.begin(), rExerciseTimes.end(),
                              std::less_equal<double>()));
  PRECONDITION(std::is_sorted(rBarrierTimes.begin(), rBarrierTimes.end(),
                              std::less_equal<double>()));

  std::vector<double>
    uEventTimes(1 + rBarrierTimes.size() + rExerciseTimes.size());
  uEventTimes.front() = rModel.initialTime();
  std::vector<double>::iterator itEnd =
    std::set_union(rBarrierTimes.begin(), rBarrierTimes.end(),
                   rExerciseTimes.begin(), rExerciseTimes.end(),
                   uEventTimes.begin()+1);
  uEventTimes.resize(itEnd-uEventTimes.begin());
  rModel.assignEventTimes(uEventTimes);

  int iTime = uEventTimes.size()-1;
  Slice uOption = rModel.cash(iTime, 0.);
  Slice uPut = rModel.cash(iTime, 0.);

  while (iTime > 0) {
    double dTime = uEventTimes[iTime];
    if (std::binary_search(rExerciseTimes.begin(),rExerciseTimes.end(),dTime)){
      uPut = max(uPut, dStrike - rModel.spot(iTime));
    }
    if (std::binary_search(rBarrierTimes.begin(),rBarrierTimes.end(),dTime)){
      uOption += indicator(rModel.spot(iTime), dUpperBarrier)*(uPut-uOption);
    }
    iTime--;
    uPut.rollback(iTime);
    uOption.rollback(iTime);
  }

  return interpolate(uOption);
}
```