```cpp
#include "Session3/Session3.hpp"

using namespace cfl;
using namespace std;

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

  std::vector<double> uEventTimes(rBarrierTimes.size() + 2);
  uEventTimes.front() = rModel.initialTime();
  copy(rBarrierTimes.begin(), rBarrierTimes.end(), uEventTimes.begin() + 1);
  uEventTimes.back() = dMaturity;
  rModel.assignEventTimes(uEventTimes);

  int iTime = uEventTimes.size() - 1;
  Slice uOption = max(rModel.spot(iTime) - dStrike, 0.);
  iTime--;
  uOption.rollback(iTime);

  while (iTime > 0)
  {
    uOption *= indicator(rModel.spot(iTime), dLowerBarrier);
    iTime--;
    uOption.rollback(iTime);
  }

  return interpolate(uOption);
}
```

```cpp
#include "Session3/Session3.hpp"

using namespace cfl;
using namespace std;

cfl::MultiFunction
prb::callOnForwardPrice(double dStrike, double dCallMaturity,
                                        double dForwardMaturity, AssetModel & rModel)
{
  PRECONDITION(rModel.initialTime() < dCallMaturity);
  PRECONDITION(dCallMaturity < dForwardMaturity);

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

  int iEventTime = 1;
  Slice uOption = max(rModel.forward(iEventTime,dForwardMaturity) - dStrike, 0.);
  uOption.rollback(0);
  return interpolate(uOption);
}
```

```cpp
#include "Session3/Session3.hpp"

using namespace cfl;
using namespace std;

cfl::MultiFunction prb::
    americanButterfly(double dPutStrike, double dCallStrike,
                      const std::vector<double> &rExerciseTimes,
                      AssetModel &rModel)
{
  PRECONDITION(dPutStrike < dCallStrike);
  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
    Slice uSpot = rModel.spot(iTime);
    uOption = max(max(uSpot - dCallStrike, dPutStrike - uSpot), uOption);
    iTime--;
    uOption.rollback(iTime);
  }
  return interpolate(uOption);
}
```

```cpp
#include "Session3/Session3.hpp"

using namespace cfl;
using namespace std;

cfl::MultiFunction prb::
corridor(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);

  int iTime = rModel.eventTimes().size()-1;
  Slice uOption = rModel.cash(iTime, 0.);
  double dMaturity = rBarrierTimes.back();
  while (iTime > 0) {
    //uOption is the value of the sum of future indicators
    uOption += (indicator(rModel.spot(iTime), dLowerBarrier)*
                indicator(dUpperBarrier, rModel.spot(iTime))
                *rModel.discount(iTime, dMaturity));
    iTime--;
    uOption.rollback(iTime);
  }
  uOption *= (dNotional/rBarrierTimes.size());
  return interpolate(uOption);
}
```