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

using namespace cfl;

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

  std::vector<double> uEventTimes(1, rModel.initialTime());
  uEventTimes.push_back(dMaturity);
  rModel.assignEventTimes(uEventTimes);

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

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

using namespace cfl;

cfl::MultiFunction
prb::americanPut(double dStrike,
                 const std::vector<double> & rExerciseTimes,
                 AssetModel & rModel)
{
  PRECONDITION(rModel.initialTime() < rExerciseTimes.front());

  std::vector<double> uEventTimes(rExerciseTimes);
  uEventTimes.insert(uEventTimes.begin(), rModel.initialTime());
  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, dStrike - rModel.spot(iTime));
    iTime--;
    uOption.rollback(iTime);
  }
  return interpolate(uOption);
}
```

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

using namespace cfl;

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

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

  int iTime = uEventTimes.size()-1;
  Slice uOption = rModel.cash(iTime, dNotional);
  while (iTime > 0) {
    //uOption is the value to continue (the value of the option
    //if no barriers have been crossed before and now)
    uOption*=indicator(rModel.spot(iTime), dLowerBarrier);
    uOption*=indicator(dUpperBarrier, rModel.spot(iTime));
    iTime--;
    uOption.rollback(iTime);
  }
  return interpolate(uOption);
}
```

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

using namespace cfl;

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

  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);
  while (iTime > 0) {
    //uOption is the value to continue
    double dTime = uEventTimes[iTime];
    if (std::binary_search(rExerciseTimes.begin(),rExerciseTimes.end(),dTime)){
      uOption = max(uOption, rModel.spot(iTime) - dStrike);
    }
    if (std::binary_search(rBarrierTimes.begin(),rBarrierTimes.end(), dTime)){
      uOption *= indicator(rModel.spot(iTime), dLowerBarrier);
    }
    iTime--;
    uOption.rollback(iTime);
  }

  return interpolate(uOption);
}
```

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

using namespace cfl;

cfl::MultiFunction
prb::swing(double dStrike,
           const std::vector<double> & rExerciseTimes,
           unsigned iNumberOfExercises, AssetModel & rModel)
{
  PRECONDITION(rModel.initialTime() < rExerciseTimes.front());

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

  int iTime = rModel.eventTimes().size()-1;
  std::vector<Slice> uOption(iNumberOfExercises, rModel.cash(iTime,0.));
  while (iTime > 0) {
    //uOption[i] is the value to continue under the condition that
    //i exercises have taken place before and at current time
    for (unsigned int iI=0; iI<uOption.size()-1; iI++) {
      uOption[iI] =
        max(uOption[iI], uOption[iI+1] + rModel.spot(iTime) - dStrike);
    }
    uOption.back() = max(uOption.back(), rModel.spot(iTime) - dStrike);
    iTime--;
    for (unsigned int iI=0; iI<uOption.size(); iI++) {
      uOption[iI].rollback(iTime);
    }
  }
  return interpolate(uOption.front());
}
```