```cpp
#include "Session1/Session1.hpp"
#include "cfl/Macros.hpp"
#include <cmath>

using namespace cfl;
using namespace cfl::Data;

class Yield : public cfl::IFunction
{
public:
  Yield(const cfl::Function &rDiscount, double dInitialTime)
      : m_uDiscount(rDiscount), m_dInitialTime(dInitialTime){}

  double operator()(double dT) const
  {
    PRECONDITION(belongs(dT));
    double dYield;
    if (dT < m_dInitialTime + cfl::EPS)
    {
      dYield = (1. - m_uDiscount(m_dInitialTime + cfl::EPS)) / cfl::EPS;
    }
    else
    {
      dYield = -std::log(m_uDiscount(dT)) / (dT - m_dInitialTime);
    }
    return dYield;
  }

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

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

cfl::Function prb::yield(const cfl::Function &rDiscount,
                         double dInitialTime)
{
  return cfl::Function(new Yield(rDiscount, dInitialTime));
}
```

```cpp
#include "Session1/Session1.hpp"

using namespace cfl;
using namespace std;
using namespace prb;

namespace NVolBlack
{
  cfl::Function yieldShape1(double dLambda, double dInitialTime)
  {
    std::function<double(double)> uShape =
        [dLambda, dInitialTime](double dT) {
          double dX = std::max(dLambda * (dT - dInitialTime), cfl::EPS);
          return (1 - std::exp(-dX)) / dX;
        };
    return Function(uShape, dInitialTime);
  }
}; // namespace NVolBlack

using namespace NVolBlack;

cfl::Function prb::
    volatilityBlack(double dSigma, double dLambda, double dInitialTime)
{
  PRECONDITION((dSigma >= 0) && (dLambda >= 0));

  return dSigma * sqrt(yieldShape1(2 * dLambda, dInitialTime));
}
```

```cpp
#include <numeric>
#include "Session1/Session1.hpp"
#include "cfl/Macros.hpp"
#include <cmath>

using namespace cfl;
using namespace std;

cfl::Function prb::yieldNelsonSiegel(double dC0, double dC1, double dC2,
                                     double dLambda, double dInitialTime)
{
  PRECONDITION(dLambda >= 0);

  std::function<double(double)> uY =
    [dC0, dC1, dC2, dLambda, dInitialTime](double dT) {
      double dX = max<double>(dLambda * (dT - dInitialTime), cfl::EPS);
      double dY = exp(-dX);
      double dZ = dC0 + dC1 * (1. - dY) / dX + dC2 * (1. - (1. + dX) * dY) / dX;
      return dZ;
    };

  return Function(uY, dInitialTime);
}
```

```cpp
#include "Session1/Session1.hpp"
#include "cfl/Macros.hpp"
#include <numeric>
#include <vector>

using namespace cfl;

cfl::Function prb::
forwardStockDividends(double dSpot,
                      std::vector<double> &rTimes,
                      std::vector<double> &rDividends,
                      const cfl::Function &rDiscount,
                      double dInitialTime)
{
  PRECONDITION(rTimes.front() > dInitialTime);
  PRECONDITION(rTimes.size() == rDividends.size());

  std::function<double(double)> uF =
    [dSpot, rTimes, rDividends, rDiscount](double dT) {
      std::vector<double>::const_iterator
        itTimeEnd = std::lower_bound(rTimes.begin(), rTimes.end(), dT);
      double dF = std::inner_product(rTimes.begin(), itTimeEnd, rDividends.begin(),
                                     dSpot, std::minus<double>(),
                                     [rDiscount](double dT, double dD) {
                                       return dD * rDiscount(dT);
                                     });
      dF /= rDiscount(dT);
      return dF;
    };
  return Function(uF, dInitialTime, rDividends.back());
}
```