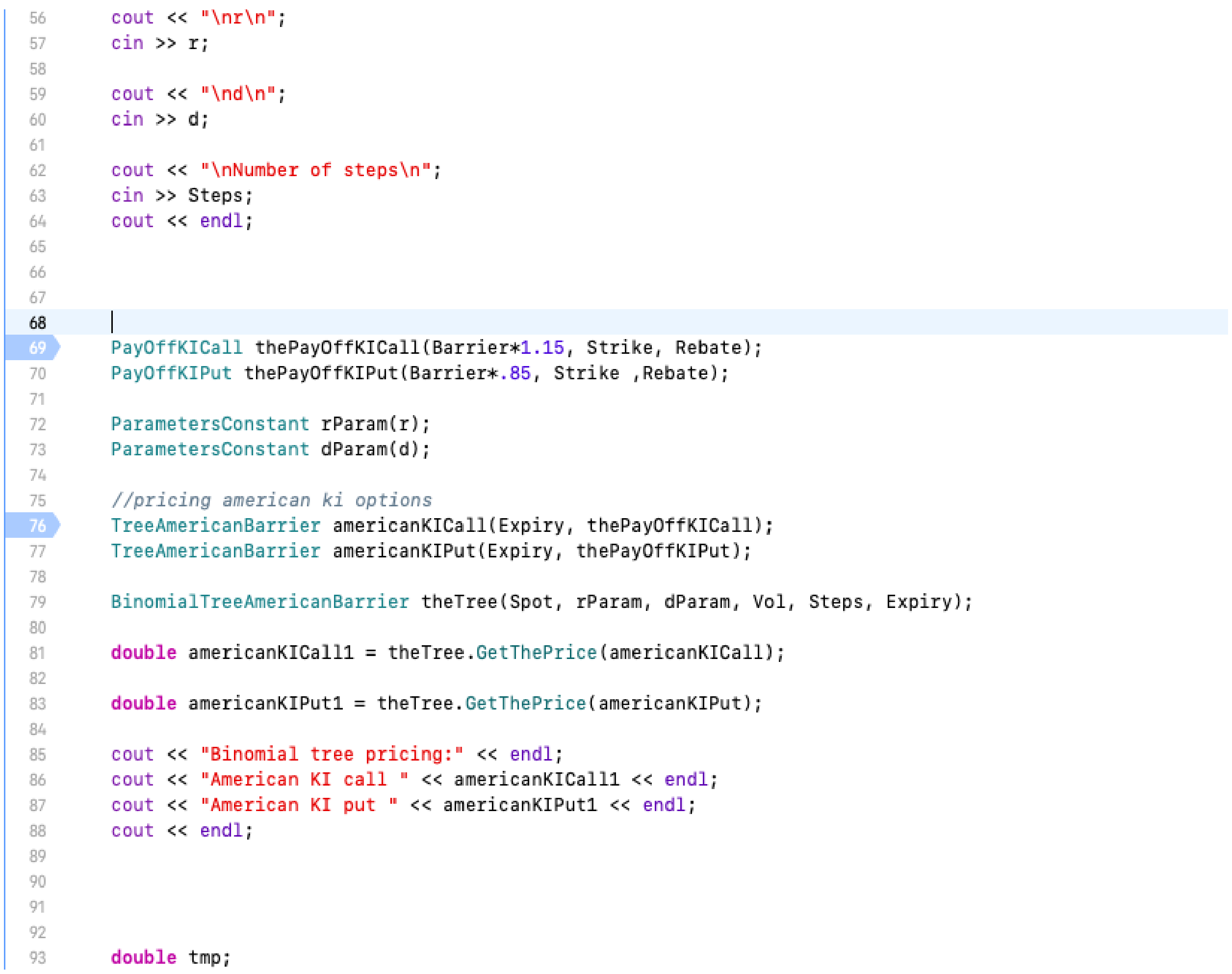
Final Project

Chapter 8 – Q-6

Solution: Starting with the main.cpp snippet .



```
cout << "\nr\n";
cin >> r;

cout << "\nd\n";
cin >> d;

cout << "\nNumber of steps\n";
cin >> Steps;
cout << endl;




PayOffKICall thePayOffKICall(Barrier*1.15, Strike, Rebate);
PayOffKIPut thePayOffKIPut(Barrier*.85, Strike ,Rebate);

ParametersConstant rParam(r);
ParametersConstant dParam(d);

//pricing american ki options
TreeAmericanBarrier americanKICall(Expiry, thePayOffKICall);
TreeAmericanBarrier americanKIPut(Expiry, thePayOffKIPut);

BinomialTreeAmericanBarrier theTree(Spot, rParam, dParam, Vol, Steps, Expiry);

double americanKICall1 = theTree.GetThePrice(americanKICall);

double americanKIPut1 = theTree.GetThePrice(americanKIPut);

cout << "Binomial tree pricing:" << endl;
cout << "American KI call " << americanKICall1 << endl;
cout << "American KI put " << americanKIPut1 << endl;
cout << endl;




double tmp;
```

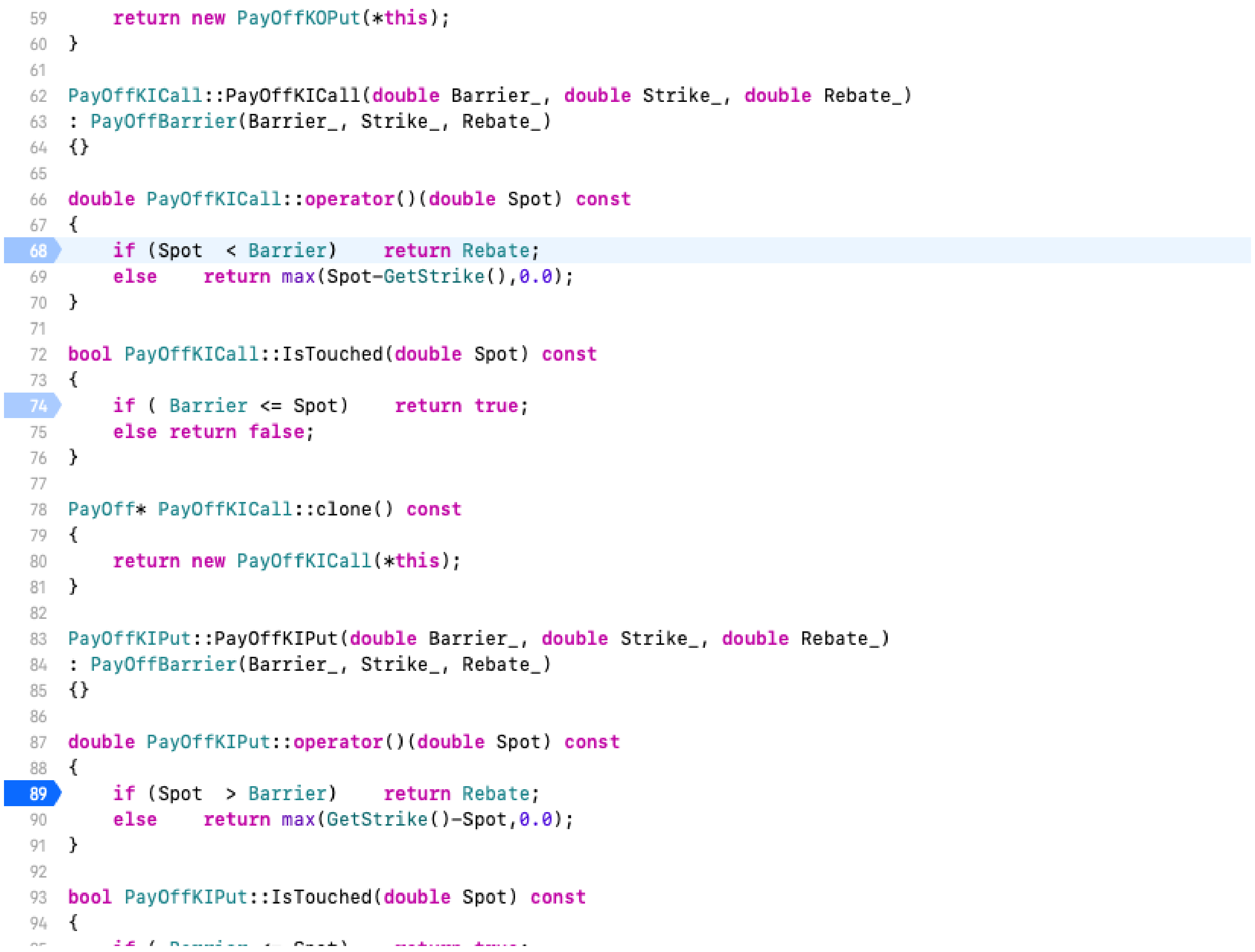
Summary of main :

1.) Instantiating the Payoff for Knock-in Call and Put options.
2.) Instantiating the TreeAmericanBarrier for both options .
3.) Instantiating the Binomial Tree for American Options.
4.) Finally calling the Get Price function of Binomial Tree.

Explanation of every module :

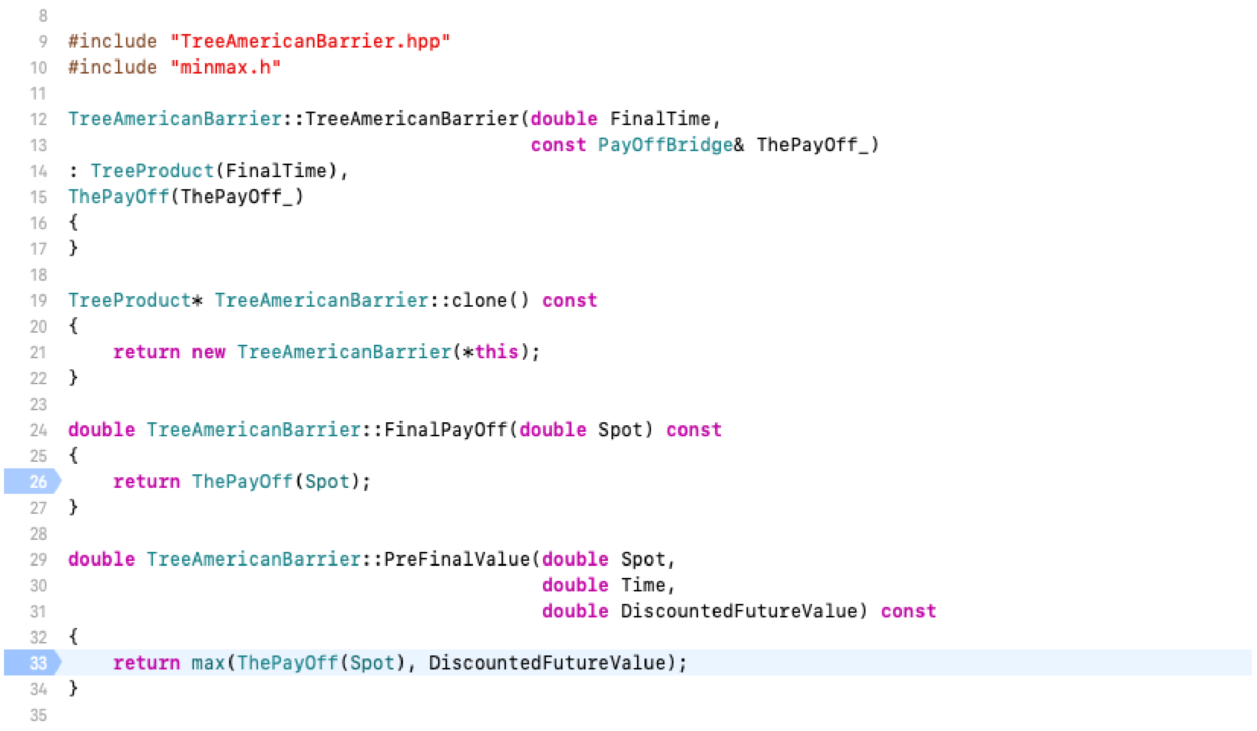
- **Knock-in Option** : Defined and declared in BarrierOption.h and .cpp files.
  Simple Logic of this Option is :
  1.) If barrier is not breached then the payoff will be some constant value Rebate.
  2.) Otherwise the Payoff will be of simple vanilla option.

Snippet is attached:



```cpp
    return new PayOffKOPut(*this);
}

PayOffKICall::PayOffKICall(double Barrier_, double Strike_, double Rebate_)
: PayOffBarrier(Barrier_, Strike_, Rebate_)
{}

double PayOffKICall::operator()(double Spot) const
{
    if (Spot  < Barrier)    return Rebate;
    else    return max(Spot-GetStrike(),0.0);
}

bool PayOffKICall::IsTouched(double Spot) const
{
    if ( Barrier <= Spot)    return true;
    else return false;
}

PayOff* PayOffKICall::clone() const
{
    return new PayOffKICall(*this);
}

PayOffKIPut::PayOffKIPut(double Barrier_, double Strike_, double Rebate_)
: PayOffBarrier(Barrier_, Strike_, Rebate_)
{}

double PayOffKIPut::operator()(double Spot) const
{
    if (Spot  > Barrier)    return Rebate;
    else    return max(GetStrike()-Spot,0.0);
}

bool PayOffKIPut::IsTouched(double Spot) const
{
    if ( Barrier <= Spot)    return true;
```

- **TreeAmericanBarrier** : Class(Derived from class TreeProduct) for calculating Final nodes payoff and payoff for other nodes specifically for American options, as the options here can be exercised earlier.
  Snippet is attached for .cpp



```cpp

#include "TreeAmericanBarrier.hpp"
#include "minmax.h"

TreeAmericanBarrier::TreeAmericanBarrier(double FinalTime,
                                         const PayOffBridge& ThePayOff_)
: TreeProduct(FinalTime),
ThePayOff(ThePayOff_)
{
}

TreeProduct* TreeAmericanBarrier::clone() const
{
    return new TreeAmericanBarrier(*this);
}

double TreeAmericanBarrier::FinalPayOff(double Spot) const
{
    return ThePayOff(Spot);
}

double TreeAmericanBarrier::PreFinalValue(double Spot,
                                          double Time,
                                          double DiscountedFutureValue) const
{
    return max(ThePayOff(Spot), DiscountedFutureValue);
}

```

Roles of this class :

1.) FinalPayoff – This will calculate or call the Knock in Barrier Payoff for last layer nodes of the Tree.
2.) PreFinalValue : This calculates max(payoff, discounted future value) for all other intermediate nodes. American Option property of early exercising comes into play here.

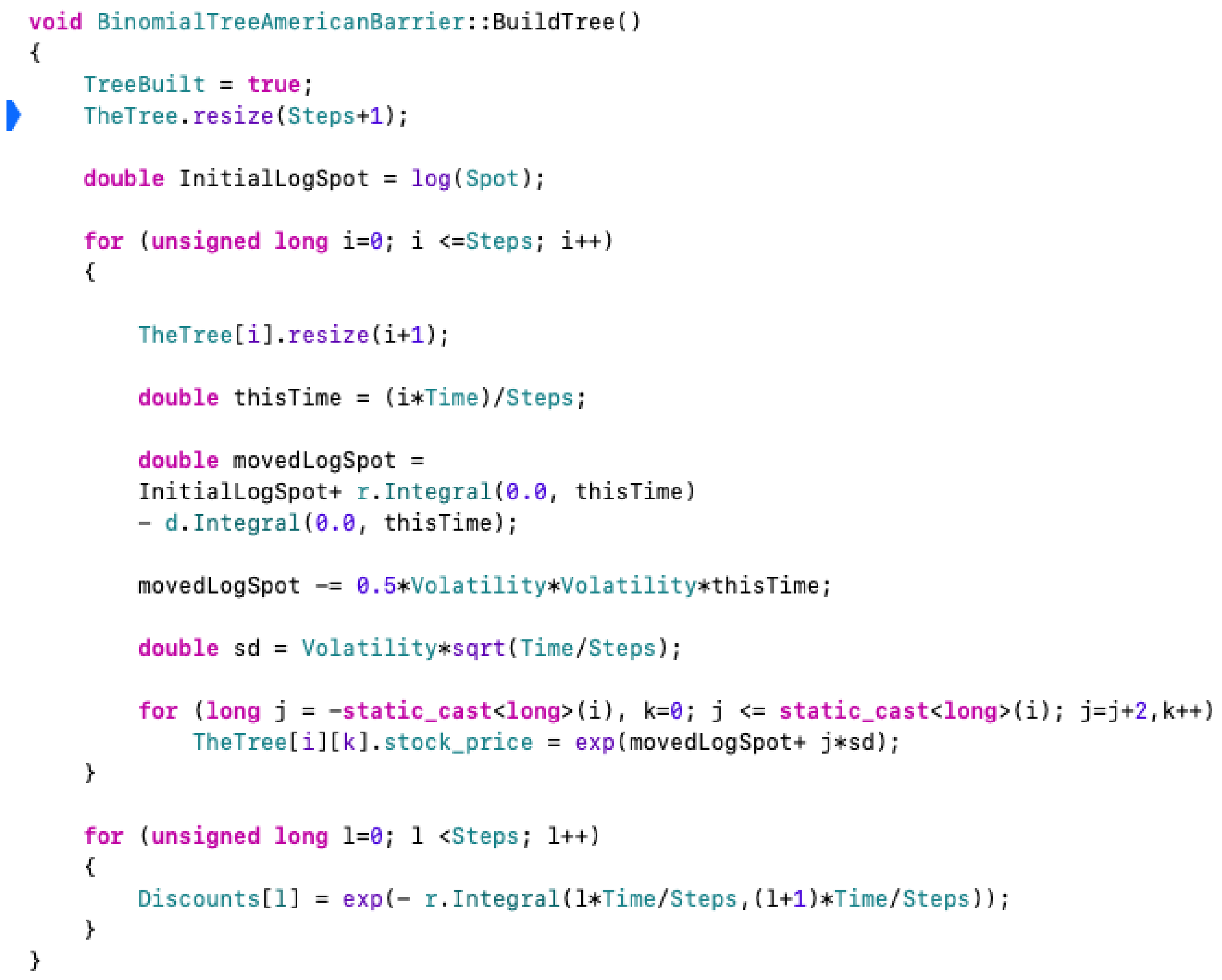
- **BinomialTreeAmericanBarrier** : Roles :

1.) The design is such that we can price multiple products with the same expiry; we call the method multiple times and only build the tree once.
2.) The method **BuildTree** creates the tree. We resize the vector describing all the layers first. We then resize each layer so that it is of the correct size.
3.) Also it computes the locations of all the nodes, and the discounts needed to compute the expectations backwards in the tree.

4.) **GetthePrice method role** : First we compute the final layer using the FinalPayOff and write the values into the second element of each pair in the final layer. After this we simply iterate backwards, computing as we go. The final value of the product is then simply the value of the option at the single node in the first layer of the tree, and that is what we return.

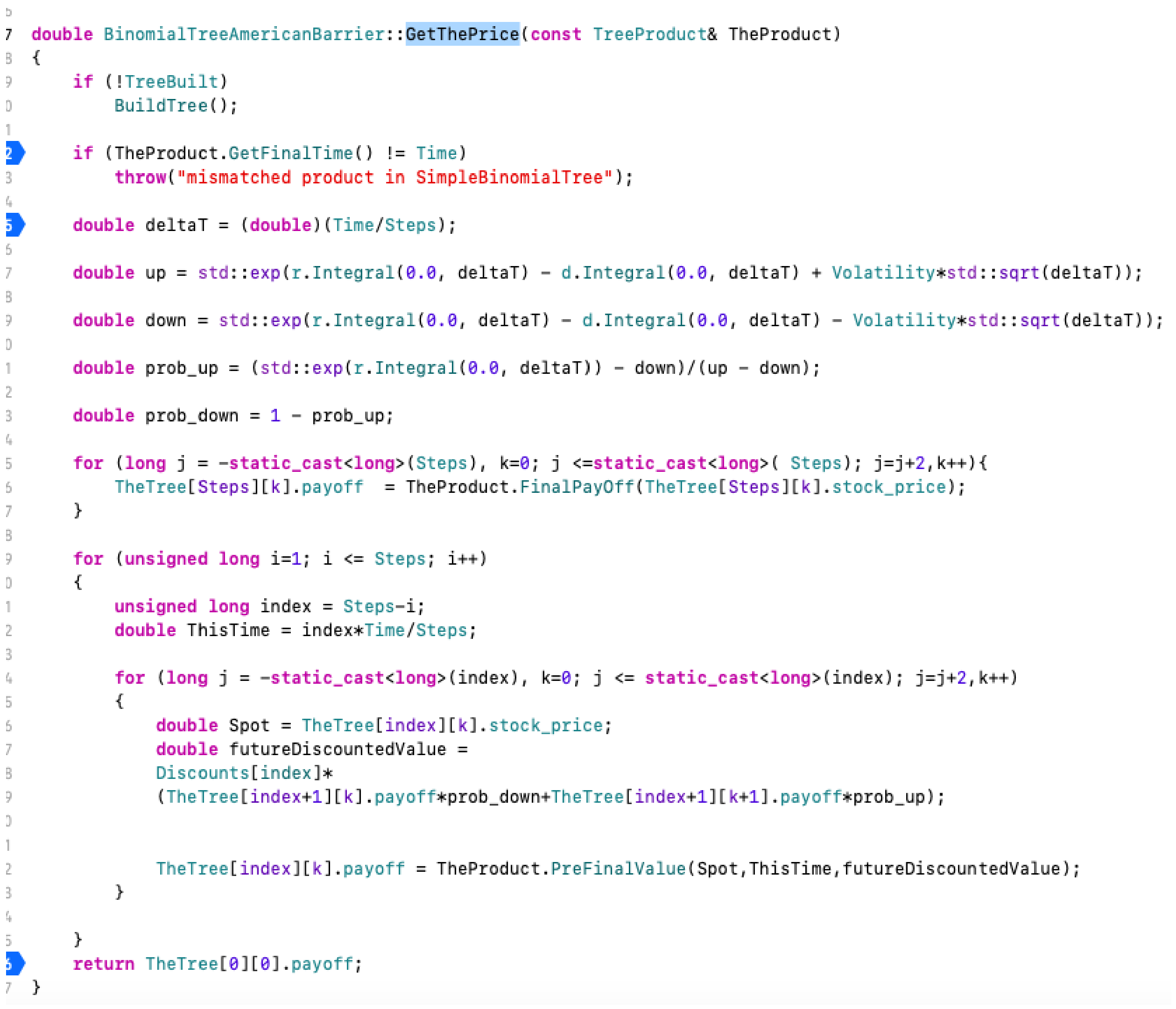
Code Snippet attached :

1.) Build Tree :



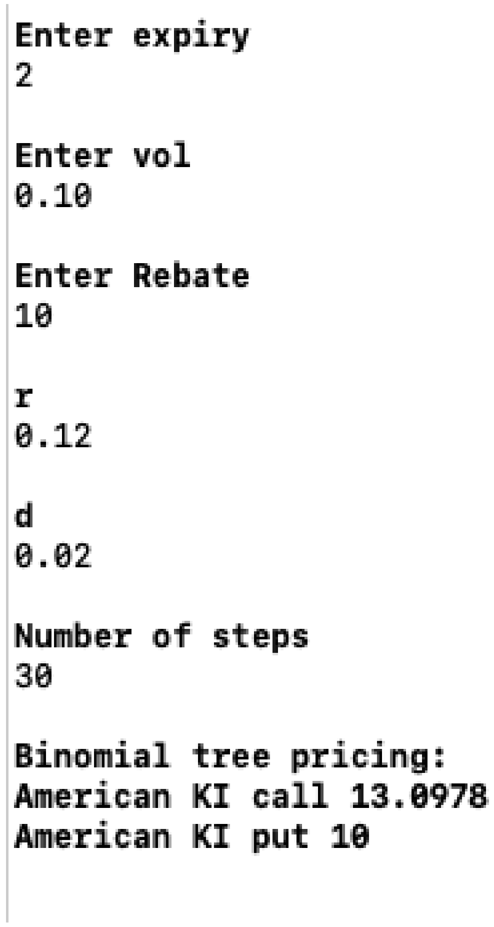
```
void BinomialTreeAmericanBarrier::BuildTree()
{
    TreeBuilt = true;
    TheTree.resize(Steps+1);

    double InitialLogSpot = log(Spot);

    for (unsigned long i=0; i <=Steps; i++)
    {

        TheTree[i].resize(i+1);

        double thisTime = (i*Time)/Steps;

        double movedLogSpot =
        InitialLogSpot+ r.Integral(0.0, thisTime)
        - d.Integral(0.0, thisTime);

        movedLogSpot -= 0.5*Volatility*Volatility*thisTime;

        double sd = Volatility*sqrt(Time/Steps);

        for (long j = -static_cast<long>(i), k=0; j <= static_cast<long>(i); j=j+2,k++)
            TheTree[i][k].stock_price = exp(movedLogSpot+ j*sd);
    }

    for (unsigned long l=0; l <Steps; l++)
    {
        Discounts[l] = exp(- r.Integral(l*Time/Steps,(l+1)*Time/Steps));
    }
}
```

2.) GetThePrice:



```cpp
double BinomialTreeAmericanBarrier::GetThePrice(const TreeProduct& TheProduct)
{
    if (!TreeBuilt)
        BuildTree();

    if (TheProduct.GetFinalTime() != Time)
        throw("mismatched product in SimpleBinomialTree");

    double deltaT = (double)(Time/Steps);

    double up = std::exp(r.Integral(0.0, deltaT) - d.Integral(0.0, deltaT) + Volatility*std::sqrt(deltaT));

    double down = std::exp(r.Integral(0.0, deltaT) - d.Integral(0.0, deltaT) - Volatility*std::sqrt(deltaT));

    double prob_up = (std::exp(r.Integral(0.0, deltaT)) - down)/(up - down);

    double prob_down = 1 - prob_up;

    for (long j = -static_cast<long>(Steps), k=0; j <=static_cast<long>( Steps); j=j+2,k++){
        TheTree[Steps][k].payoff  = TheProduct.FinalPayOff(TheTree[Steps][k].stock_price);
    }

    for (unsigned long i=1; i <= Steps; i++)
    {
        unsigned long index = Steps-i;
        double ThisTime = index*Time/Steps;

        for (long j = -static_cast<long>(index), k=0; j <= static_cast<long>(index); j=j+2,k++)
        {
            double Spot = TheTree[index][k].stock_price;
            double futureDiscountedValue =
            Discounts[index]*
            (TheTree[index+1][k].payoff*prob_down+TheTree[index+1][k+1].payoff*prob_up);


            TheTree[index][k].payoff = TheProduct.PreFinalValue(Spot,ThisTime,futureDiscountedValue);
        }

    }
    return TheTree[0][0].payoff;
}
```

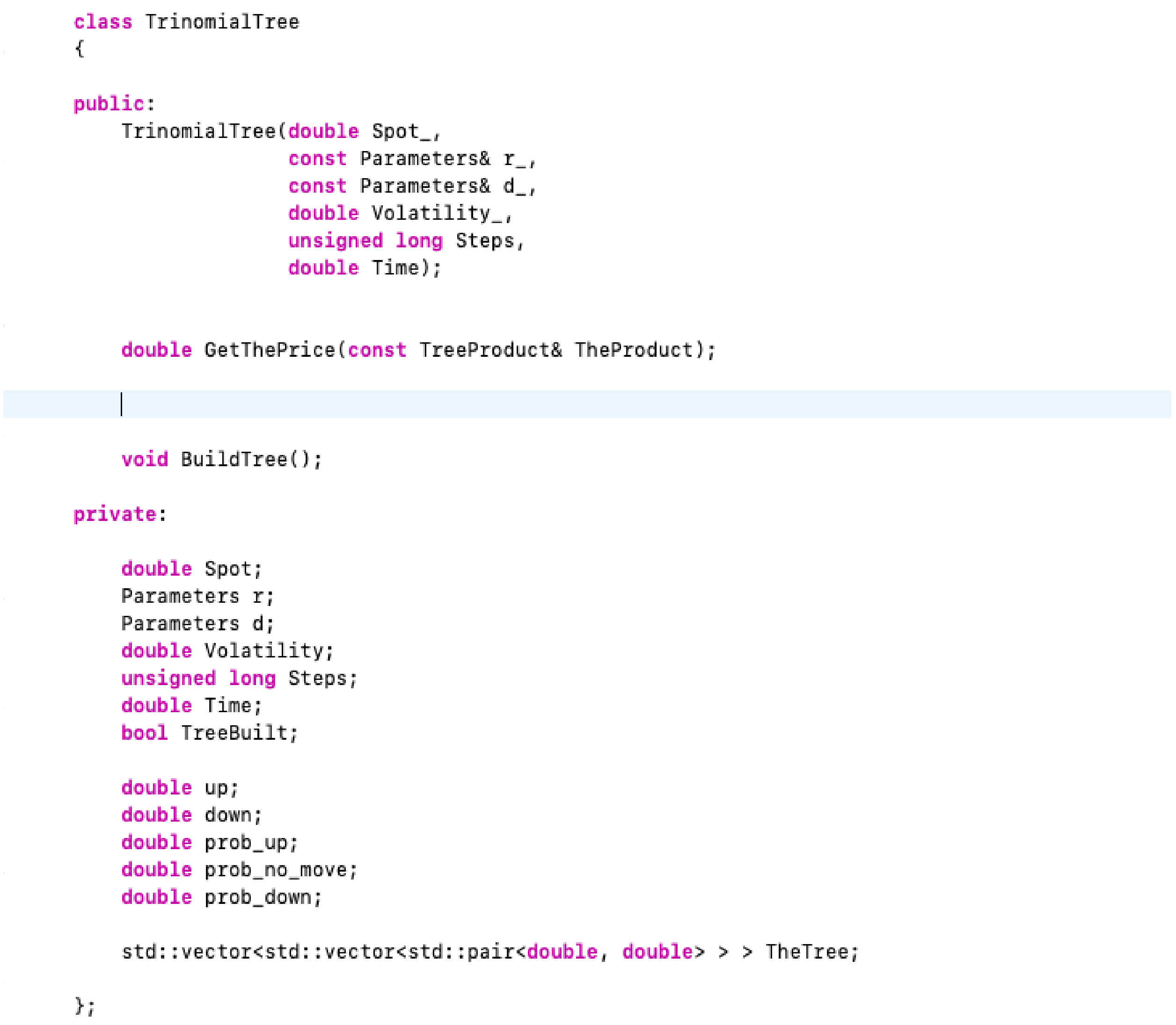
Output :



```
Enter expiry
2

Enter vol
0.10

Enter Rebate
10

r
0.12

d
0.02

Number of steps
30

Binomial tree pricing:
American KI call 13.0978
American KI put 10
```

Bonus Problem of Trinomial Tree :

Header File :



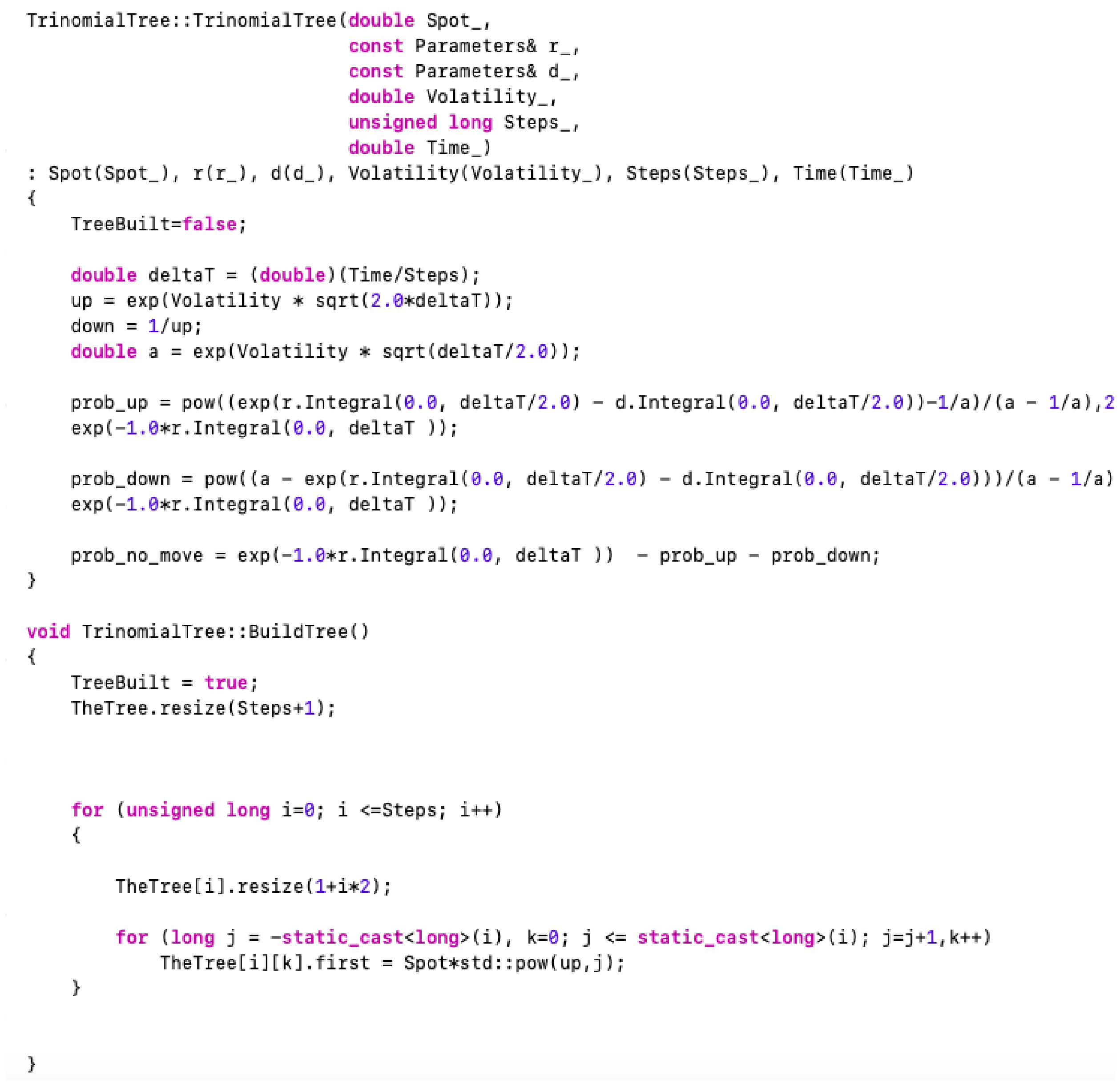
```
class TrinomialTree
{

public:
    TrinomialTree(double Spot_,
                  const Parameters& r_,
                  const Parameters& d_,
                  double Volatility_,
                  unsigned long Steps,
                  double Time);


    double GetThePrice(const TreeProduct& TheProduct);



    void BuildTree();

private:

    double Spot;
    Parameters r;
    Parameters d;
    double Volatility;
    unsigned long Steps;
    double Time;
    bool TreeBuilt;

    double up;
    double down;
    double prob_up;
    double prob_no_move;
    double prob_down;

    std::vector<std::vector<std::pair<double, double> > > TheTree;

};
```

1.) Based on the Trinomial Tree property adding one more extra probability term for no_move.
2.) Because of one extra term the Build and Get the price method will change too.

.CPP File

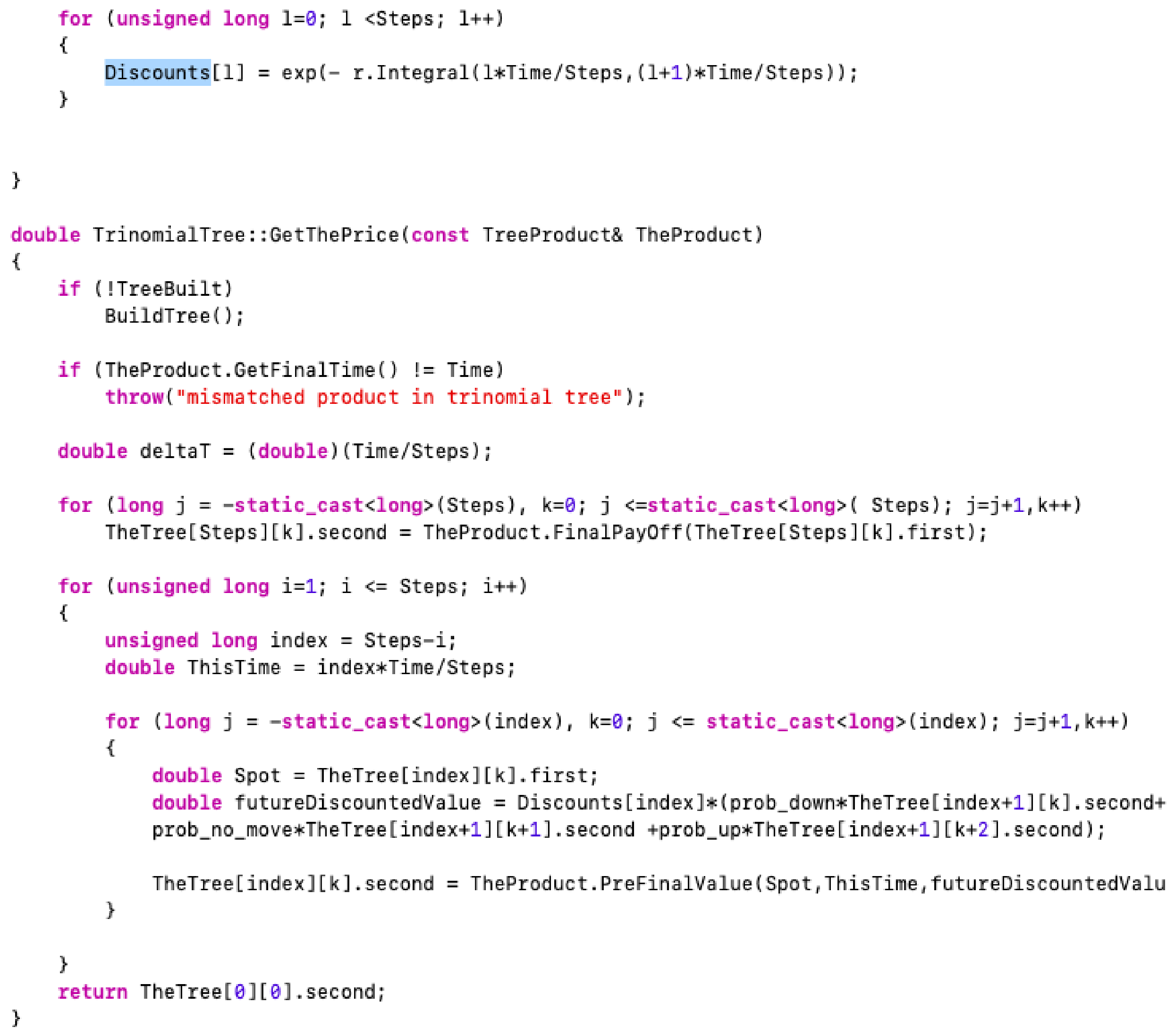
Constructor and BuildTree function :

1.) Here the probability calculation function for up, down and no_movement has changed.
2.) Logic for number of nodes in a level is also increased.



```
TrinomialTree::TrinomialTree(double Spot_,
                             const Parameters& r_,
                             const Parameters& d_,
                             double Volatility_,
                             unsigned long Steps_,
                             double Time_)
: Spot(Spot_), r(r_), d(d_), Volatility(Volatility_), Steps(Steps_), Time(Time_)
{
    TreeBuilt=false;

    double deltaT = (double)(Time/Steps);
    up = exp(Volatility * sqrt(2.0*deltaT));
    down = 1/up;
    double a = exp(Volatility * sqrt(deltaT/2.0));

    prob_up = pow((exp(r.Integral(0.0, deltaT/2.0) - d.Integral(0.0, deltaT/2.0))-1/a)/(a - 1/a),2
    exp(-1.0*r.Integral(0.0, deltaT ));

    prob_down = pow((a - exp(r.Integral(0.0, deltaT/2.0) - d.Integral(0.0, deltaT/2.0)))/(a - 1/a)
    exp(-1.0*r.Integral(0.0, deltaT ));

    prob_no_move = exp(-1.0*r.Integral(0.0, deltaT ))  - prob_up - prob_down;
}

void TrinomialTree::BuildTree()
{
    TreeBuilt = true;
    TheTree.resize(Steps+1);



    for (unsigned long i=0; i <=Steps; i++)
    {

        TheTree[i].resize(1+i*2);

        for (long j = -static_cast<long>(i), k=0; j <= static_cast<long>(i); j=j+1,k++)
            TheTree[i][k].first = Spot*std::pow(up,j);
    }


}
```

GetthePrice function of TrinomialTree :
Here the only change is FutureDiscountvalue calculation which uses the new probabilities .



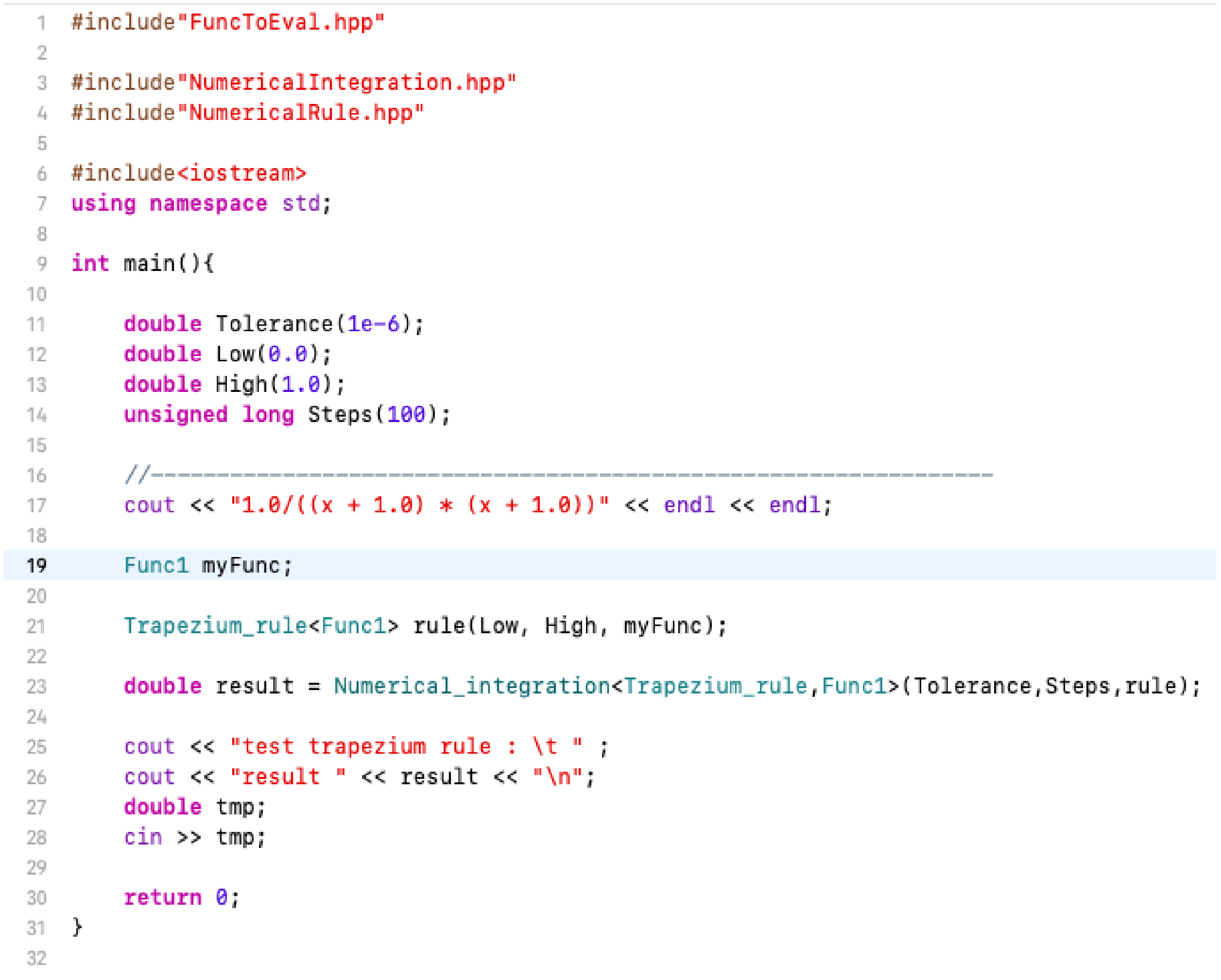
```
    for (unsigned long l=0; l <Steps; l++)
    {
        Discounts[l] = exp(- r.Integral(l*Time/Steps,(l+1)*Time/Steps));
    }


}

double TrinomialTree::GetThePrice(const TreeProduct& TheProduct)
{
    if (!TreeBuilt)
        BuildTree();

    if (TheProduct.GetFinalTime() != Time)
        throw("mismatched product in trinomial tree");

    double deltaT = (double)(Time/Steps);

    for (long j = -static_cast<long>(Steps), k=0; j <=static_cast<long>( Steps); j=j+1,k++)
        TheTree[Steps][k].second = TheProduct.FinalPayOff(TheTree[Steps][k].first);

    for (unsigned long i=1; i <= Steps; i++)
    {
        unsigned long index = Steps-i;
        double ThisTime = index*Time/Steps;

        for (long j = -static_cast<long>(index), k=0; j <= static_cast<long>(index); j=j+1,k++)
        {
            double Spot = TheTree[index][k].first;
            double futureDiscountedValue = Discounts[index]*(prob_down*TheTree[index+1][k].second+
            prob_no_move*TheTree[index+1][k+1].second +prob_up*TheTree[index+1][k+2].second);

            TheTree[index][k].second = TheProduct.PreFinalValue(Spot,ThisTime,futureDiscountedValu
        }

    }
    return TheTree[0][0].second;
}
```

In conclusion only the new class is added for TrinomialTree apart from that all the pattern will remain same.

Q2 – Chapter 9 – Q-2

Solution : The whole exercise was to solve a integration with a Trapezoid method, since the function to take the integration of can be anything, that's why using templatization.

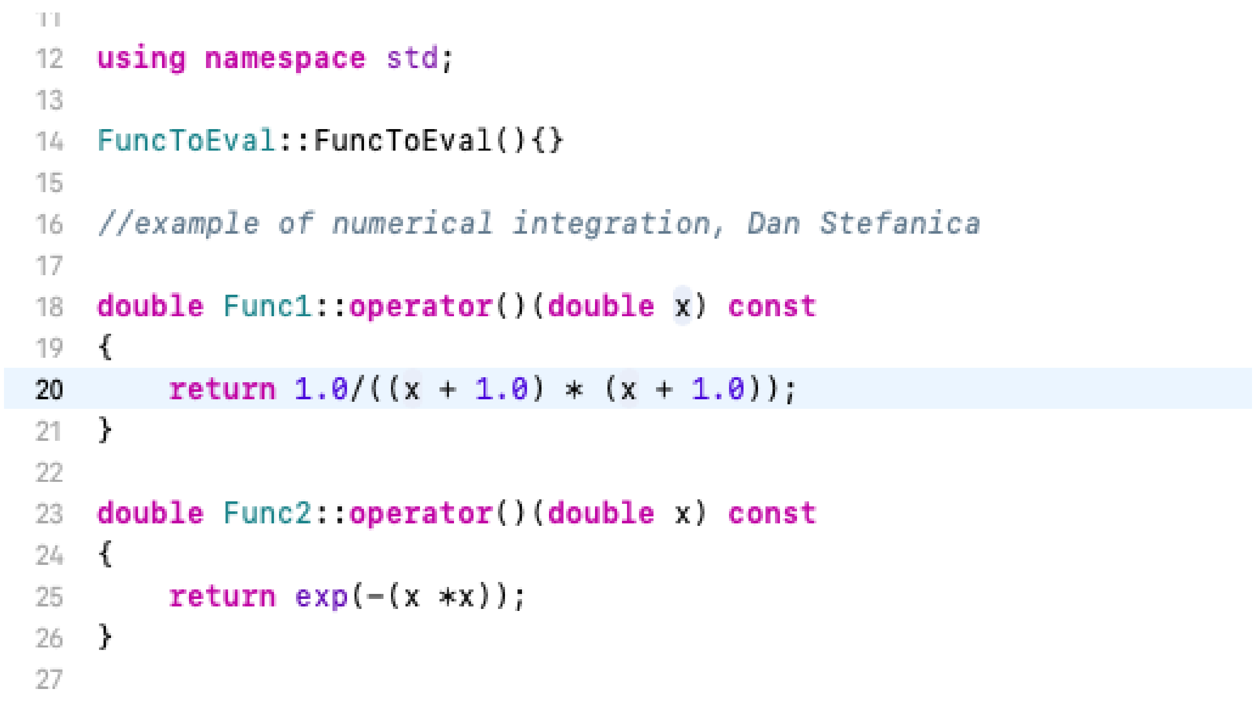
Snippet of main.cpp



```cpp
#include"FuncToEval.hpp"

#include"NumericalIntegration.hpp"
#include"NumericalRule.hpp"

#include<iostream>
using namespace std;

int main(){

    double Tolerance(1e-6);
    double Low(0.0);
    double High(1.0);
    unsigned long Steps(100);

    //-------------------------------------------------------------------
    cout << "1.0/((x + 1.0) * (x + 1.0))" << endl << endl;

    Func1 myFunc;

    Trapezium_rule<Func1> rule(Low, High, myFunc);

    double result = Numerical_integration<Trapezium_rule,Func1>(Tolerance,Steps,rule);

    cout << "test trapezium rule : \t " ;
    cout << "result " << result << "\n";
    double tmp;
    cin >> tmp;

    return 0;
}

```

Explanation :

1.) **Func1** : This class contains all the functions you want to take integration of,
Snippet :



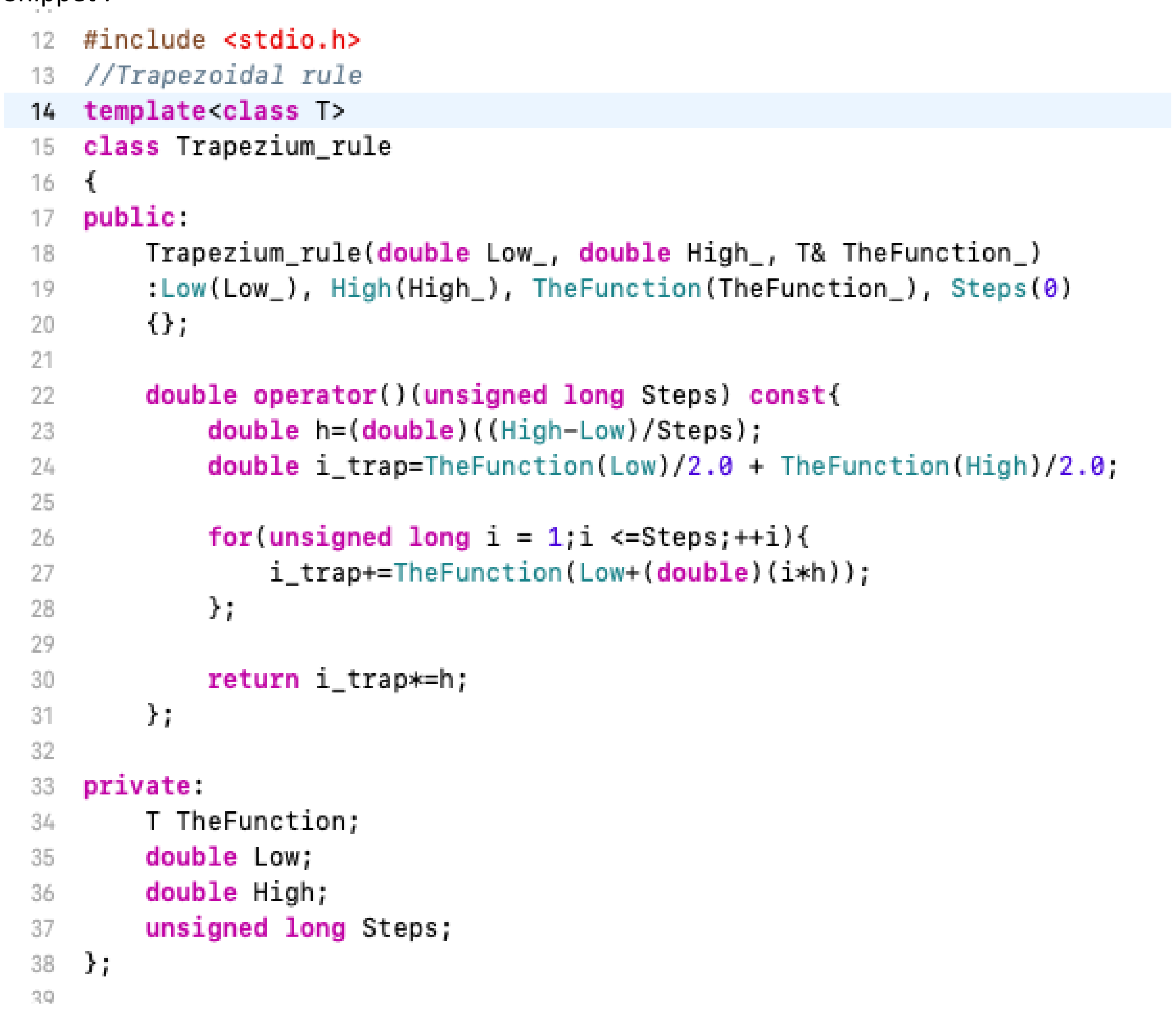
```
using namespace std;

FuncToEval::FuncToEval(){}

//example of numerical integration, Dan Stefanica

double Func1::operator()(double x) const
{
    return 1.0/((x + 1.0) * (x + 1.0));
}

double Func2::operator()(double x) const
{
    return exp(-(x *x));
}

```

2.) **TrapeziumRule** : Estimates the integration by using this pseudocode :

```
a = left endpoint of the integration interval
b = right endpoint of the integration interval
n = number of partition intervals
f_int(x) = routine evaluating f(x)

output:
I_trap = trapezoidal rule approximation of integral(f(x),a,b)
h = (b -a)/n;
```

```
I_trap = f_int(a)/2 + f_int(b)/2
for i = 1:n-1
        I_trap = I_trap+ f_int(a + i*h)
end
I_trap = h * I_trap
```

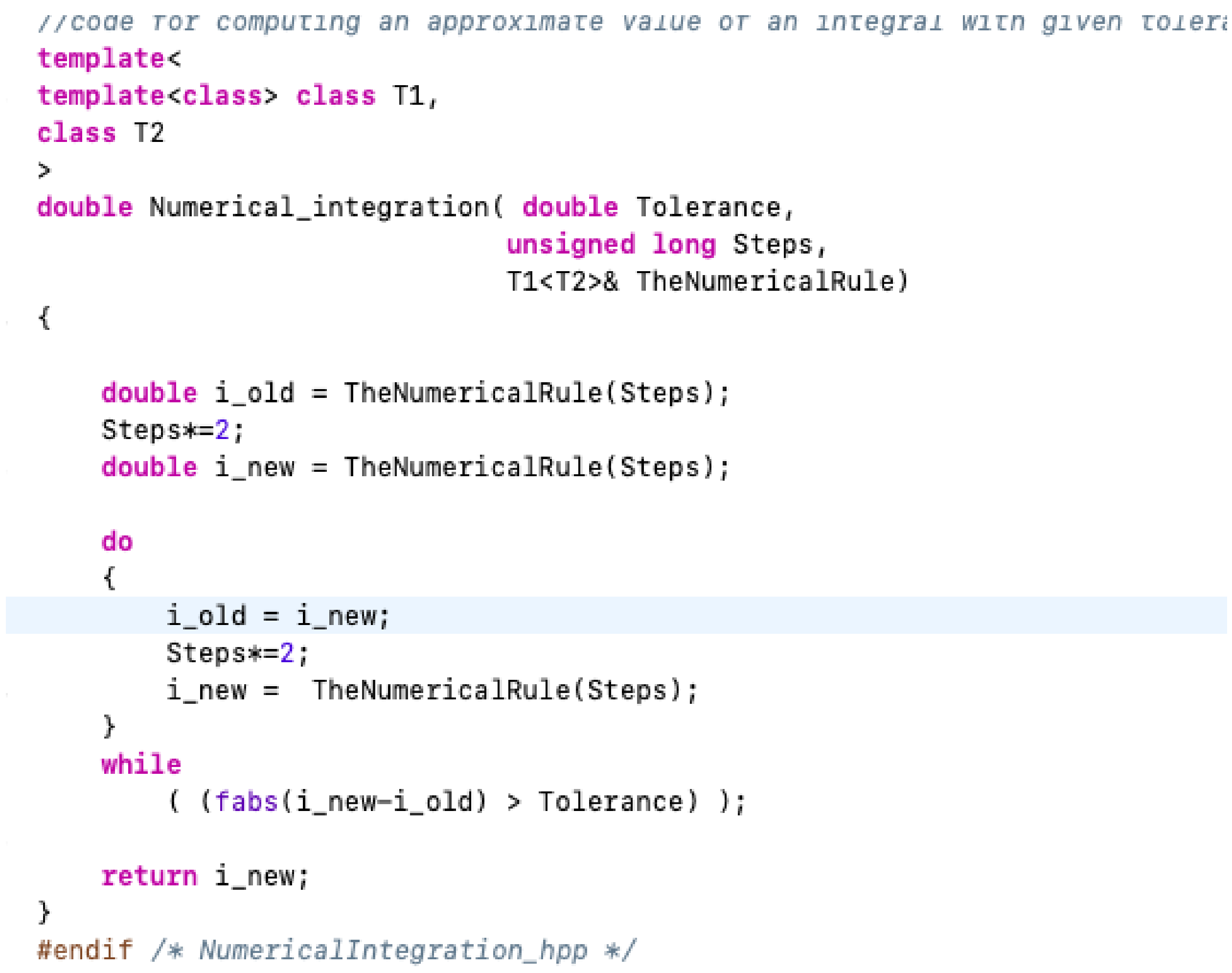
Snippet :



```cpp
#include <stdio.h>
//Trapezoidal rule
template<class T>
class Trapezium_rule
{
public:
    Trapezium_rule(double Low_, double High_, T& TheFunction_)
    :Low(Low_), High(High_), TheFunction(TheFunction_), Steps(0)
    {};

    double operator()(unsigned long Steps) const{
        double h=(double)((High-Low)/Steps);
        double i_trap=TheFunction(Low)/2.0 + TheFunction(High)/2.0;

        for(unsigned long i = 1;i <=Steps;++i){
            i_trap+=TheFunction(Low+(double)(i*h));
        };

        return i_trap*=h;
    };

private:
    T TheFunction;
    double Low;
    double High;
    unsigned long Steps;
};
```

Use of Templatization here : The basic idea of **templatization** is that you can write code that works for many classes simultaneously provided they are required to have certain operations defined with the same syntax. And here the operator function acts on all those function objects.

3.) **NumericalIntegration** : Role of this class is to compute the Trapezoidal method with increasing number of steps, such that the computations start converging the before-mentioned Tolerance level.



```
//code for computing an approximate value of an integral with given tolera
template<
template<class> class T1,
class T2
>
double Numerical_integration( double Tolerance,
                              unsigned long Steps,
                              T1<T2>& TheNumericalRule)
{

    double i_old = TheNumericalRule(Steps);
    Steps*=2;
    double i_new = TheNumericalRule(Steps);

    do
    {
        i_old = i_new;
        Steps*=2;
        i_new =  TheNumericalRule(Steps);
    }
    while
        ( (fabs(i_new-i_old) > Tolerance) );

    return i_new;
}
#endif /* NumericalIntegration_hpp */
```

Output :

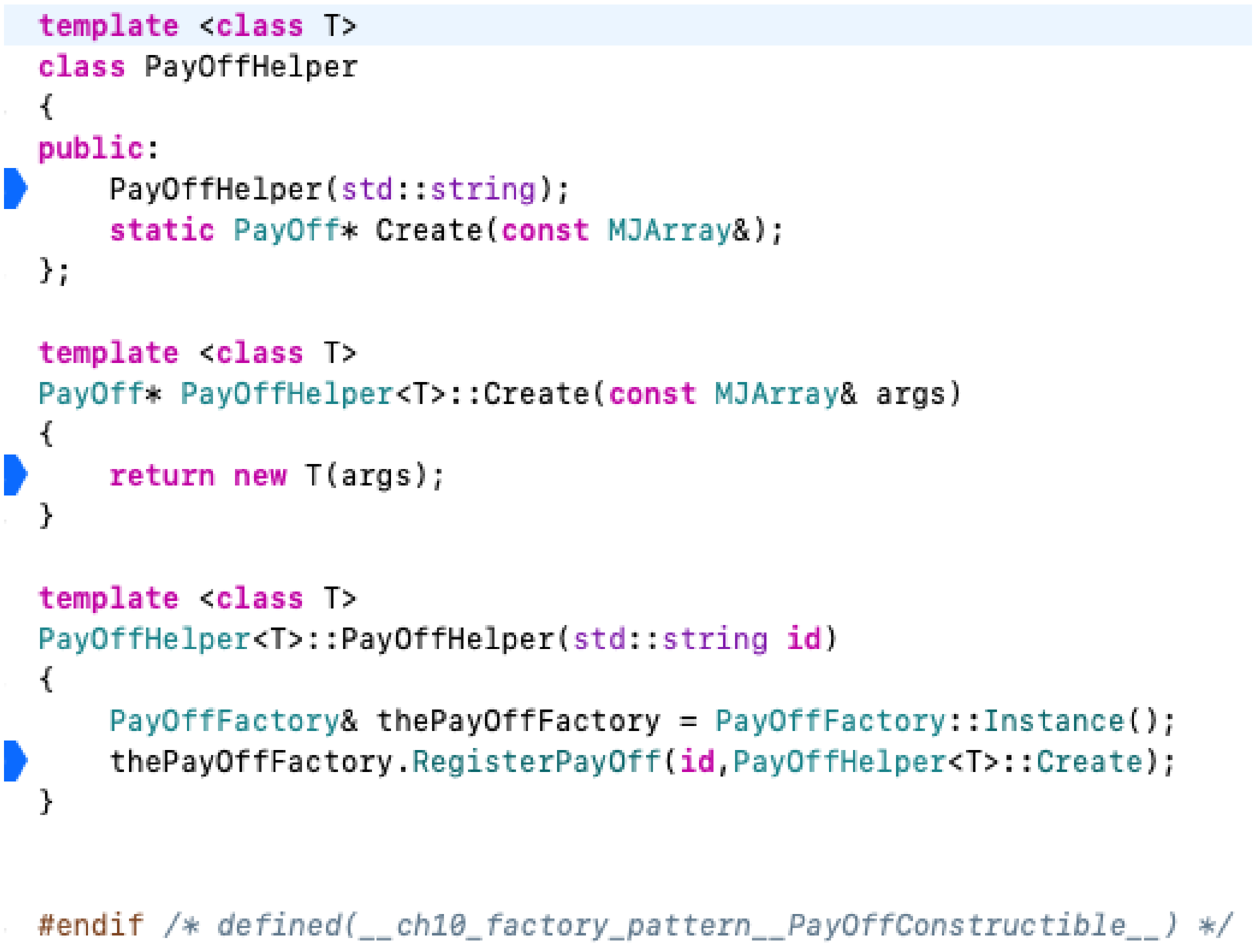
```
test trapezium rule :      result 0.632121
```

Q3 – Ch-9 q-2

Solutions : Factory Pattern for DoubleDigitOption with two arguments. I used Mjarray args instead of Strike. Here args can take any number of arguments.
Requirements for Factory pattern :

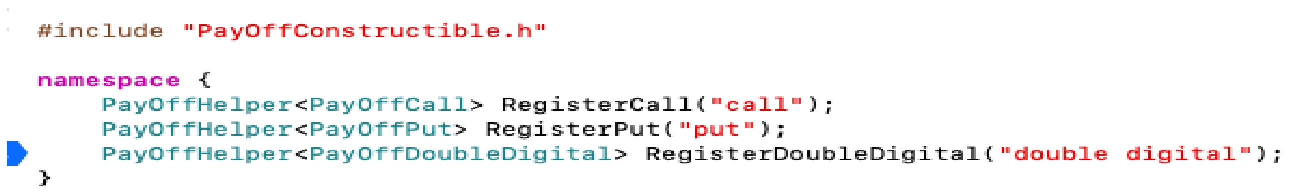
1.) Get the class to communicate with the factory, without explicitly calling anything from the main routine. Therefore using Global variables.
2.) Using Templatization wrote a helper class whose constructor registers the payoff class with our factory.

Snippet – Helper class :



```
template <class T>
class PayOffHelper
{
public:
    PayOffHelper(std::string);
    static PayOff* Create(const MJArray&);
};

template <class T>
PayOff* PayOffHelper<T>::Create(const MJArray& args)
{
    return new T(args);
}

template <class T>
PayOffHelper<T>::PayOffHelper(std::string id)
{
    PayOffFactory& thePayOffFactory = PayOffFactory::Instance();
    thePayOffFactory.RegisterPayOff(id,PayOffHelper<T>::Create);
}


#endif /* defined(__ch10_factory_pattern__PayOffConstructible__) */
```

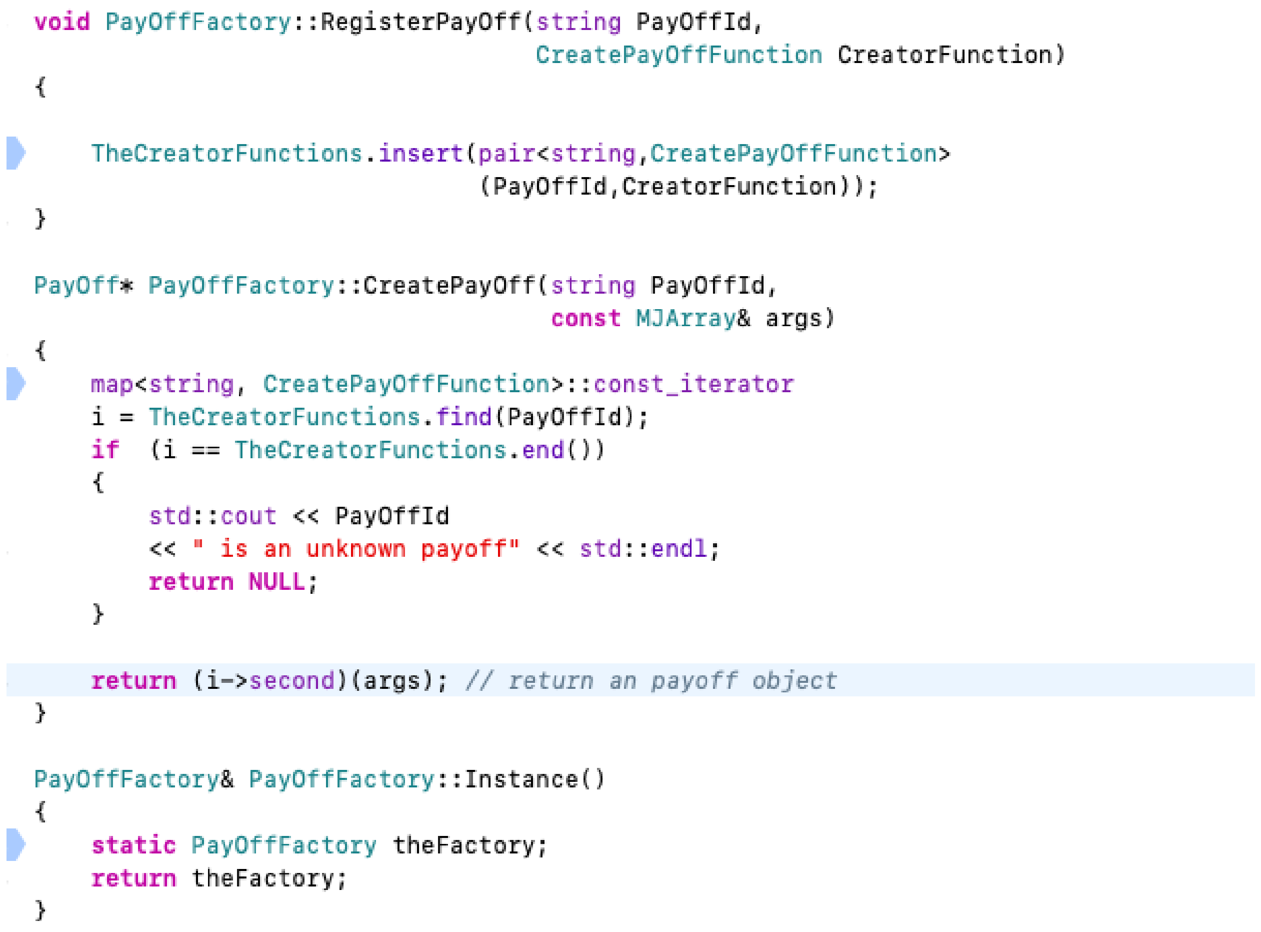
Global Variables :



```
#include "PayOffConstructible.h"

namespace {
     PayOffHelper<PayOffCall> RegisterCall("call");
     PayOffHelper<PayOffPut> RegisterPut("put");
     PayOffHelper<PayOffDoubleDigital> RegisterDoubleDigital("double digital");
}
```

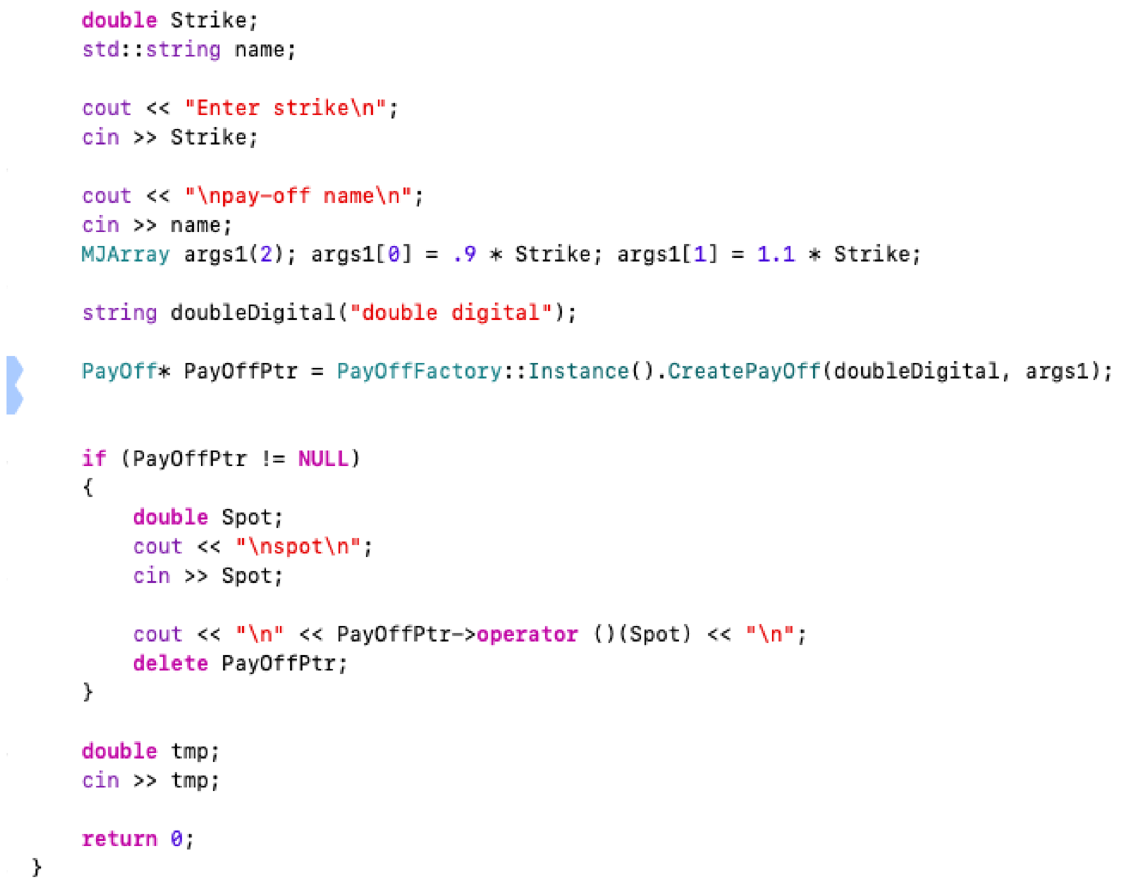
Global Variables are initialized at the start of the program. This initialization carries out the registration as required.

3.) Payoff Factory Class Snippet with MjArray args instead of single Strike.



```
void PayOffFactory::RegisterPayOff(string PayOffId,
                                   CreatePayOffFunction CreatorFunction)
{

    TheCreatorFunctions.insert(pair<string,CreatePayOffFunction>
                               (PayOffId,CreatorFunction));
}

PayOff* PayOffFactory::CreatePayOff(string PayOffId,
                                    const MJArray& args)
{
    map<string, CreatePayOffFunction>::const_iterator
    i = TheCreatorFunctions.find(PayOffId);
    if  (i == TheCreatorFunctions.end())
    {
        std::cout << PayOffId
        << " is an unknown payoff" << std::endl;
        return NULL;
    }

    return (i->second)(args); // return an payoff object
}

PayOffFactory& PayOffFactory::Instance()
{
    static PayOffFactory theFactory;
    return theFactory;
}
```

4.) Main File taking MJarray args :



```
    double Strike;
    std::string name;

    cout << "Enter strike\n";
    cin >> Strike;

    cout << "\npay-off name\n";
    cin >> name;
    MJArray args1(2); args1[0] = .9 * Strike; args1[1] = 1.1 * Strike;

    string doubleDigital("double digital");

    PayOff* PayOffPtr = PayOffFactory::Instance().CreatePayOff(doubleDigital, args1);


    if (PayOffPtr != NULL)
    {
        double Spot;
        cout << "\nspot\n";
        cin >> Spot;

        cout << "\n" << PayOffPtr->operator ()(Spot) << "\n";
        delete PayOffPtr;
    }

    double tmp;
    cin >> tmp;

    return 0;
}
```