# Topic 6:
# Class Templates – Price Tree & Stopping Tree for American Options

## Class Templates

- We would like to compute and store the price of an American option not only at time 0, but also for each time step n and node i in the binomial tree.
- In addition, we want to compute the **early exercise policy** for an American option. The time step n and node i at which the option should be exercised are characterized by the condition:
  - $H(n, i) = h(S(n, i)) > 0$

- The nature structure for the price data is a lattice index by the time steps n = 0, 1, …., N and nodes i = 0, 1, …, n.
- A convenient way is to store the option prices in a vector indexed by the time variable ***n*** consisting vectors of type ***double*** indexed by the nodes ***i*** at each time ***n***:

```
vector< vector<double> > Lattice;
void SetN(int N_)
    { N=N_;
      Lattice.resize(N+1);
      for(int n=0; n<=N; n++) Lattice[n].resize(n+1);
    }
```

## BinLattice01.h

```
#pragma once
#include <vector>
#include <iomanip>
using namespace std;
namespace fre {
   class BinLattice
   {
   private:
      int N;
      vector< vector<double> > Lattice;
   public:
      BinLattice():N(0)
      {}
      BinLattice(int N_) :N(N_)
      {
         Lattice.resize(N + 1);
         for (int n = 0; n <= N; n++) Lattice[n].resize(n + 1);
      }
      ~BinLattice()
      { }
```

## BinLattice01.h (Continue)

```
void SetN(int N_)
{   N = N_;
    Lattice.resize(N + 1);
    for (int n = 0; n <= N; n++) Lattice[n].resize(n + 1);
}
void SetNode(int n, int i, double x)
{   Lattice[n][i] = x;              }
double GetNode(int n, int i)
{    return Lattice[n][i];        }
void Display()
{   cout << setiosflags(ios::fixed) << setprecision(3);
    for (int n = 0; n <= N; n++)
    {
        for (int i = 0; i <= n; i++)
            cout << setw(15) << GetNode(n, i);
        cout << endl;
    }
    cout << endl;
}
  };
}
```

## Notes:

- The ***BinLattice*** contains two variables and four functions:
  - ***N*** to store the number of time steps in the binomial tree.
  - ***Lattice***, a vector of vector to hold data of type double.
  - The ***SetN()*** function takes a parameter of type int, assigns it to ***N***, and sets the size of the ***Lattice*** vector of ***N+1***, the number of time instants ***n*** from 0 to N, and then for each n sets the size of the inner vector ***Lattice[n]*** to ***n+1***, the number of nodes at time ***n.***
  - ***SetNode()*** to set the value stored at step ***n***, node ***i***.
  - ***GetNode()*** to return the value stored at step ***n***, node ***i***.
  - ***Display()*** to print the value stored at step ***n***, node ***i***.
- The entire code for the ***BinLattice*** class can be found in the header file, ***BinLattice01.h.*** The reason is that BinLattice class is going to converted into a class template, which does not lend itself to separate compilation.

# Option07.h

```
#pragma once
#include "BinomialTreeModel02.h"
#include "BinLattice01.h"
namespace fre {
    class Option
    {
    private:
      Option() : N(0) {}
      Option(const Option& option) : N(option.N) {}
    protected:
      int N;
    public:
      Option(int N_) : N(N_) {}
      int GetN() const { return N; }
      virtual double Payoff(double z) const = 0;
      virtual ~Option() = 0;
    };
```

# Option07.h (Continue)

```
class Call : public Option
{
private:
  double K;
public:
  Call(int N_, double K_) : Option(N_), K(K_) {}
  ~Call() {}
  double Payoff(double z) const;
};

class Put : public Option
{
private:
  double K;
public:
  Put(int N_, double K_) : Option(N_), K(K_) {}
  ~Put() {}
  double Payoff(double z) const;
};
```

# Option07.h (Continue)

```
class OptionCalculation
{
private:
  Option* pOption;
  OptionCalculation() : pOption(0) {}
  OptionCalculation(const OptionCalculation& optionCalculation) :
pOption(optionCalculation.pOption) {}
public:
  OptionCalculation(Option* pOption_) : pOption(pOption_) {}
  ~OptionCalculation() {}
  double PriceByCRR(const BinomialTreeModel& Model);
  double PriceBySnell(const BinomialTreeModel& Model, BinLattice & PriceTree);
};
}
```

## Option07.cpp

```
#pragma once
#include "Option07.h"
#include "BinomialTreeModel02.h"
#include "BinLattice01.h"
#include <iostream>
#include <cmath>
using namespace std;
namespace fre {
   Option::~Option()
   {    cout << "Option Destructor" << endl;    }
   double Call::Payoff(double z) const
   {
      if (z > K) return z - K;
      return 0.0;
   }
   double Put::Payoff(double z) const
   {
      if (z < K) return K - z;
      return 0.0;
   }
```

## Option07.cpp (Continue)

```
double OptionCalculation::PriceByCRR(const BinomialTreeModel& Model)
{
   double q = Model.RiskNeutProb();
   int N = pOption->GetN();
   vector<double> Price(N + 1);
   for (int i = 0; i <= N; i++)
   {
      Price[i] = pOption->Payoff(Model.CalculateAssetPrice(N, i));
   }
   for (int n = N - 1; n >= 0; n--)
   {
      for (int i = 0; i <= n; i++)
      {
         Price[i] = (q * Price[i + 1] + (1 - q) * Price[i]) / Model.GetR();
      }
   }
   return Price[0];
}
```

# Option07.cpp (Continue)

```
double OptionCalculation::PriceBySnell(const BinomialTreeModel& Model, BinLattice & PriceTree)
{   double q = Model.RiskNeutProb();
    int N = pOption->GetN();
    double ContVal = 0;
    for (int i = 0; i <= N; i++)
    {
        PriceTree.SetNode(N, i, pOption->Payoff(Model.CalculateAssetPrice(N, i)));
    }
    for (int n = N - 1; n >= 0; n--)
    {   for (int i = 0; i <= n; i++)
        {   ContVal = (q * PriceTree.GetNode(n + 1, i + 1) + (1 - q) * PriceTree.GetNode(n + 1, i)) / Model.GetR();
            PriceTree.SetNode(n, i, pOption->Payoff(Model.CalculateAssetPrice(n, i)));
            if (ContVal > PriceTree.GetNode(n, i))
            {
                PriceTree.SetNode(n, i, ContVal);
            }
        }
    }
    return PriceTree.GetNode(0, 0);
}
}
```

## OptionPricer07.cpp

```
#include "BinomialTreeModel02.h"
#include "Option07.h"
#include <iostream>
#include <iomanip>
using namespace std;
using namespace fre;
int main()
{     int N = 8;
      double U = 1.15125, D = 0.86862, R = 1.00545;
      double S0 = 106.00, K = 100.00;
      BinomialTreeModel Model(S0, U, D, R);
      Call call(N, K);
      OptionCalculation callCalculation(&call);
      cout << "European call option price = "
        << fixed << setprecision(2) << callCalculation.PriceByCRR(Model) << endl;
      Put put(N, K);
      OptionCalculation putCalculation(&put);
      cout << "European put option price = "
        << fixed << setprecision(2) << putCalculation.PriceByCRR(Model) << endl;
```

## OptionPricer07.cpp (Continue)

```
    BinLattice CallPriceTree(N);
    cout << "American call option price = "
      << fixed << setprecision(2) << callCalculation.PriceBySnell(Model, CallPriceTree) << endl;

    cout << "American call price tree:" << endl << endl;
    CallPriceTree.Display();

    BinLattice PutPriceTree(N);
    cout << "American put option price = "
      << fixed << setprecision(2) << putCalculation.PriceBySnell(Model, PutPriceTree) << endl;

    cout << "American put price tree:" << endl << endl;
    PutPriceTree.Display();

    return 0;
}
```

# OptionPricer07.cpp (Continue)

```
/*
BinomialTreeModel Constructor with Parameters S0 =106 U = 1.15125 D = 0.86862 R = 1.00545
European call option price = 21.68
European put option price = 11.43
American call option price = 21.68
American call price tree:

  21.681
  12.057   32.180
   5.574   19.101   46.479
   1.875    9.578   29.464   65.132
   0.322    3.551   16.107   44.028   88.353
   0.000    0.670    6.661   26.354   63.356  115.982
   0.000    0.000    1.391   12.352   41.571   87.283  147.869
   0.000    0.000    0.000    2.889   22.574   62.281  114.907  184.657
   0.000    0.000    0.000    0.000    5.999   40.489   86.202  146.788  227.087
```

# OptionPricer07.cpp (Continue)

```
American put option price = 11.72
American put price tree:

      11.724
      16.734      6.517
      23.161     10.075      2.799
      30.930     15.143      4.788      0.712
      39.657     21.978      8.030      1.388      0.000
      47.585     30.530     13.113      2.704      0.000      0.000
      54.471     39.657     20.611      5.271      0.000      0.000      0.000
      60.453     47.585     30.530     10.273      0.000      0.000      0.000      0.000
      65.649     54.471     39.657     20.023      0.000      0.000      0.000      0.000      0.000

*/
```

## Class Template for BinLattice class

- To record the stopping policy, it would be easy to write a separate class for the data of type ***bool*** by following the pattern of the ***BinLattice*** class. However, we don't want the headache of several duplicate code fragment to maintain.
- ***Class Templates*** offer a much neat solution: The type is not hardwired inside the class, but passed to it as a parameter:
  - ***template<typename Type> class BinLattice***
  - A class template with type parameter type.

# C++ Function Templates

- Approaches for functions that implement identical tasks for different data types
  - Naive Approach
  - Function Overloading
  - Function Template
- Instantiating a Function Templates

# Approach 1: Naive Approach

- Create unique functions with unique names for each combination of data types
  - Difficult to keep track of multiple function names
  - lead to programming errors

## Example

```
void PrintInt( int n )
{
    cout << "***Printing:" << endl;
    cout << "Value is " << n << endl;
}
void PrintChar( char ch )
{
    cout << "***Printing:" << endl;
    cout << "Value is " << ch << endl;
}
void PrintFloat( float x )
{
    cout << "***Printing:" << endl;
    cout << "Value is " << x << endl;
}
```

**To output the traced values, we insert:**

```
PrintInt(sum);

PrintChar(initial);

PrintFloat(angle);
```

## Approach 2: Function Overloading – Review

- The use of the same name for different C++ functions, distinguished from each other by their parameter lists
  - Eliminates the need to come up with many different names for identical tasks.
  - Reduces the chance of unexpected results caused by using the wrong function name.

# Example of Function Overloading

```
void Print( int n )
{
    cout << "***Printing:" << endl;
    cout << "Value is " << n << endl;
}
void Print( char ch )
{
    cout << "***Printing" << endl;
    cout << "Value is " << ch << endl;
}
void Print( float x )
{
    cout << "***Printing:" << endl;
    cout << "Value is " << x << endl;
}
```

**To output the traced values, we insert:**

```
Print(someInt);
Print(someChar);
Print(someFloat);
```

# Approach 3: Function Template

- A C++ language construct that allows the compiler to generate multiple versions of a function by allowing parameterized data types.

**FunctionTemplate**

```
template < TemplateParamList >
FunctionDefinition
```

**TemplateParamDeclaration: placeholder**

```
class      typeIdentifier
typename variableIdentifier
```

# Example of a Function Template

```
template<class SomeType>

void Print( SomeType val )
{
    cout << "***Printing:" << endl;
    cout << "Value is " << val << endl;
}
```

*Template parameter*

*(class, user defined type, built-in types)*

*Template argument*

To output the traced values, we insert:

```
Print<int>(sum);
Print<char>(initial);
Print<float>(angle);
```

# Instantiating a Function Template

- When the compiler instantiates a template, it substitutes the template argument for the template parameter throughout the function template.

**TemplateFunction Call**

**Function < TemplateArgList > (FunctionArgList)**

# Summary of Three Approaches

**Naive Approach**
Different Function Definitions
Different Function Names

**Function Overloading**
Different Function Definitions
Same Function Name

**Template Functions**
One Function Definition (a function template)
Compiler Generates Individual Functions

# Class Template

- A C++ language construct that allows the compiler to generate multiple versions of a class by allowing parameterized data types.

**Class Template**

```
template < TemplateParamList >
ClassDefinition
```

**TemplateParamDeclaration: placeholder**

```
{  class     typeIdentifier
   typename variableIdentifier
```

# Example of a Class Template

```
template <class ItemType>
class GList
{
 public:
    bool IsEmpty() const;
    bool IsFull() const;
    int  Length() const;
    void Insert( ItemType item );
    void Delete( ItemType item );
    bool IsPresent( ItemType item ) const;
    void Sort();
    void Print() const;
    GList();                        // Constructor
 private:
    int      length;
    ItemType data[MAX_LENGTH];
};
```

*Template parameter*

# Instantiating a Class Template

- Class template arguments must be explicit.
- The compiler generates distinct class types called *template classes* or *generated classes*.
- When instantiating a template, a compiler substitutes the template argument for the template parameter throughout the class template.

# Instantiating a Class Template

To create lists of different data types

```
// Client code

GList<int> list1;
GList<float> list2;
GList<string> list3;

list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");
```

***template argument***

Compiler generates 3 distinct class types

```
GList_int list1;
GList_float list2;
GList_string list3;
```

# Substitution Example

```
class GList_int
{
public:                                      int

           void Insert(ItemType item );
                                                 int
           void Delete( ItemType item );

           bool IsPresent(ItemType item ) const;

private:                                        int
           int       length;
           ItemType data[MAX_LENGTH];
};
                 int
```

# Function Definitions for Members of a Template Class

```
template <class ItemType>
void GList<ItemType>::Insert( ItemType item )
{
    data[length] = item;
    length++;
}
```

```
//after substitution of float
void GList<float>::Insert( float item )
{
    data[length] = item;
    length++;
}
```

# Another Template Example: passing two parameters

```
template <class T, int size>
  class Stack {...
    T buf[size];
  };
Stack<int,128> mystack;
```

non-type parameter

# BinLattice02.h

```
#pragma once
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
namespace fre {
   template<typename Type>
   class BinLattice
   {
   private:
      int N;
      vector< vector<Type> > Lattice;
   public:
      BinLattice() : N(0) {}
      BinLattice(int N_) :N(N_)
      {   Lattice.resize(N + 1);
         for (int n = 0; n <= N; n++) Lattice[n].resize(n + 1);
      }
      ~BinLattice() {}
```

# BinLattice02.h (continue)

```
    void SetN(int N_)
    {  N = N_;
       Lattice.resize(N + 1);
       for (int n = 0; n <= N; n++) Lattice[n].resize(n + 1);
    }
    void SetNode(int n, int i, Type x)
    {   Lattice[n][i] = x;          }
    Type GetNode(int n, int i)
    {   return Lattice[n][i];       }
    void Display()
    {  cout << setiosflags(ios::fixed) << setprecision(3);
       for (int n = 0; n <= N; n++)
       {  for (int i = 0; i <= n; i++)
             cout << setw(15) << GetNode(n, i);
          cout << endl;
       }
       cout << endl;
    }
  };
}
```

## Notes:

- vector < vector<***double***> > Lattice is replaced by
  - vector< vector<***Type***> > Lattice;
- The member function, void SetNode(int n, int i, ***double*** x) is replaced by:
  - void SetNode(int n, int i, ***Type*** x)
- The member function, ***double*** GetNode(int n, int i) is repleaced by:
  - ***Type*** GetNode(int n, int i)
- There is no .cpp file corresponding to BinLattice02.h. A class template can only be compiled after an object has been declared using the template with a specific data type, for example, ***double***, substituted for the type parameter.

# Option08.h

```
#pragma once
#include "BinomialTreeModel02.h"
#include "BinLattice02.h"
namespace fre {
    class Option
    {
    private:
        Option() : N(0) {}
        Option(const Option& option) : N(option.N) {}
    protected:
        int N;
    public:
        Option(int N_) : N(N_) {}
        int GetN() const { return N; }
        virtual double Payoff(double z) const = 0;
        virtual ~Option() = 0;
    };
```

# Option08.h (continue)

```
class Call : public Option
{
private:
        double K;
public:
        Call(int N_, double K_) : Option(N_), K(K_) {}
        ~Call() {}
        double Payoff(double z) const;
};

class Put : public Option
{
private:
        double K;
public:
        Put(int N_, double K_) : Option(N_), K(K_) {}
        ~Put() {}
        double Payoff(double z) const;
};
```

# Option08.h (continue)

```
class OptionCalculation
{
private:
      Option* pOption;
      OptionCalculation() : pOption(0) {}
      OptionCalculation(const OptionCalculation& optionCalculation) :
                  pOption(optionCalculation.pOption) {}
public:
      OptionCalculation(Option* pOption_) : pOption(pOption_) {}
      ~OptionCalculation() {}
      double PriceByCRR(const BinomialTreeModel& Model);
      double PriceBySnell(const BinomialTreeModel& Model,
                  BinLattice<double>& PriceTree,
                  BinLattice<bool>& StoppingTree);
};
}
```

# Option08.cpp

```
#pragma once
#include "Option08.h"
#include "BinomialTreeModel02.h"
#include "BinLattice02.h"
#include <iostream>
#include <cmath>
using namespace std;
namespace fre {
   Option::~Option() {}
   double Call::Payoff(double z) const
   {
      if (z > K) return z - K;
      return 0.0;
   }
   double Put::Payoff(double z) const
   {
      if (z < K) return K - z;
      return 0.0;
   }
```

# Option08.cpp (Continue)

```
double OptionCalculation::PriceByCRR(const BinomialTreeModel& Model)
{
   double q = Model.RiskNeutProb();
   int N = pOption->GetN();
   vector<double> Price(N + 1);
   for (int i = 0; i <= N; i++)
   {
      Price[i] = pOption->Payoff(Model.CalculateAssetPrice(N, i));
   }
   for (int n = N - 1; n >= 0; n--)
   {
      for (int i = 0; i <= n; i++)
      {
         Price[i] = (q * Price[i + 1] + (1 - q) * Price[i]) / Model.GetR();
      }
   }
   return Price[0];
}
```

# Option08.cpp (Continue)

```
double OptionCalculation::PriceBySnell(const BinomialTreeModel& Model,
                    BinLattice<double>& PriceTree, BinLattice<bool>& StoppingTree)
{  double q = Model.RiskNeutProb();
   int N = pOption->GetN();
   PriceTree.SetN(N);
   StoppingTree.SetN(N);
   double ContVal = 0;
   for (int i = 0; i <= N; i++)
   {   PriceTree.SetNode(N, i, pOption->Payoff(Model.CalculateAssetPrice(N, i)));
       StoppingTree.SetNode(N, i, 1);
   }
```

# Option08.cpp (Continue)

```
   for (int n = N - 1; n >= 0; n--)
   {   for (int i = 0; i <= n; i++)
       {
          ContVal = (q * PriceTree.GetNode(n + 1, i + 1) + (1 - q) * PriceTree.GetNode(n + 1, i)) / Model.GetR();
          PriceTree.SetNode(n, i, pOption->Payoff(Model.CalculateAssetPrice(n, i)));
          StoppingTree.SetNode(n, i, 1);
          if (ContVal > PriceTree.GetNode(n, i))
          {
              PriceTree.SetNode(n, i, ContVal);
              StoppingTree.SetNode(n, i, 0);
          }
          else if (PriceTree.GetNode(n, i) == 0.0)
          {
              StoppingTree.SetNode(n, i, 0);
          }
       }
   }
   return PriceTree.GetNode(0, 0);
}
}
```

# Option08.cpp (Continue)

```
   for (int n = N - 1; n >= 0; n--)
   {  for (int i = 0; i <= n; i++)
      {
        ContVal = (q * PriceTree.GetNode(n + 1, i + 1) + (1 - q) * PriceTree.GetNode(n + 1, i)) / Model.GetR();
        PriceTree.SetNode(n, i, pOption->Payoff(Model.CalculateAssetPrice(n, i)));
        StoppingTree.SetNode(n, i, 1);
        if (ContVal > PriceTree.GetNode(n, i))
        {
           PriceTree.SetNode(n, i, ContVal);
           StoppingTree.SetNode(n, i, 0);
        }
        else if (PriceTree.GetNode(n, i) == 0.0)
        {
           StoppingTree.SetNode(n, i, 0);
        }
      }
   }
   return PriceTree.GetNode(0, 0);
 }
}
```

## Notes:

- Two new objects ***PriceTree*** and ***StoppingTree*** are passed by reference to ***PriceBySnell().*** We want the values computed and placed at the nodes to remain available after the function terminates.
  - ***PriceTree*** is an object of ***BinLattice<double>*** class.
  - ***StoppingTree*** is an object of ***BinLattice<bool>*** class.
  - The compiler can generate different classese from the BinLattice<> template. This is achieved by substituting specific type names for the type parameter within the angular brackets <>.
- The size of the ***PriceTree*** and ***StoppingTree*** is determined by the number of steps to expiry for the option.

# OptionPricer08.cpp

```
#include "BinomialTreeModel02.h"
#include "Option08.h"
#include <iostream>
#include <iomanip>
using namespace std;
using namespace fre;
int main()
{       int N = 8;
        double U = 1.15125, D = 0.86862, R = 1.00545;
        double S0 = 106.00, K = 100.00;
        BinomialTreeModel Model(S0, U, D, R);
        Call call(N, K);
        OptionCalculation callCalculation(&call);
        cout << "European call option price = "
          << fixed << setprecision(3) << callCalculation.PriceByCRR(Model) << endl;
        Put put(N, K);
        OptionCalculation putCalculation(&put);
        cout << "European put option price = "
          << fixed << setprecision(3) << putCalculation.PriceByCRR(Model) << endl;
```

# OptionPricer08.cpp (continue)

```
    BinLattice<double> CallPriceTree(N);
    BinLattice<bool> CallStoppingTree(N);
    cout << "American call option price = " << fixed << setprecision(3) <<
                callCalculation.PriceBySnell(Model, CallPriceTree, CallStoppingTree) << endl;
    cout << "American call price tree:" << endl << endl;
    CallPriceTree.Display();
    cout << "American call exercise policy:" << endl << endl;
    CallStoppingTree.Display();

    BinLattice<double> PutPriceTree(N);
    BinLattice<bool> PutStoppingTree(N);
    cout << "American put option price = "  << fixed << setprecision(3) <<
                putCalculation.PriceBySnell(Model, PutPriceTree, PutStoppingTree) << endl;
    cout << "American put price tree:" << endl << endl;
    PutPriceTree.Display();
    cout << "American put exercise policy:" << endl << endl;
    PutStoppingTree.Display();
    return 0;
}
```

# OptionPricer08.cpp (continue)

```
/* European call option price = 21.68
European put option price = 11.43
American call option price = 21.68
American call price tree:
        21.681
        12.057     32.180
         5.574     19.101     46.479
         1.875      9.578     29.464     65.132
         0.322      3.551     16.107     44.028     88.353
         0.000      0.670      6.661     26.354     63.356    115.982
         0.000      0.000      1.391     12.352     41.571     87.283    147.869
         0.000      0.000      0.000      2.889     22.574     62.281    114.907    184.657
         0.000      0.000      0.000      0.000      5.999     40.489     86.202    146.788    227.087
American call exercise policy:
         0
         0       0
         0       0       0
         0       0       0       0
         0       0       0       0       0
         0       0       0       0       0       0
         0       0       0       0       0       0       0
         0       0       0       0       0       0       0       0
         1       1       1       1       1       1       1       1       1
```

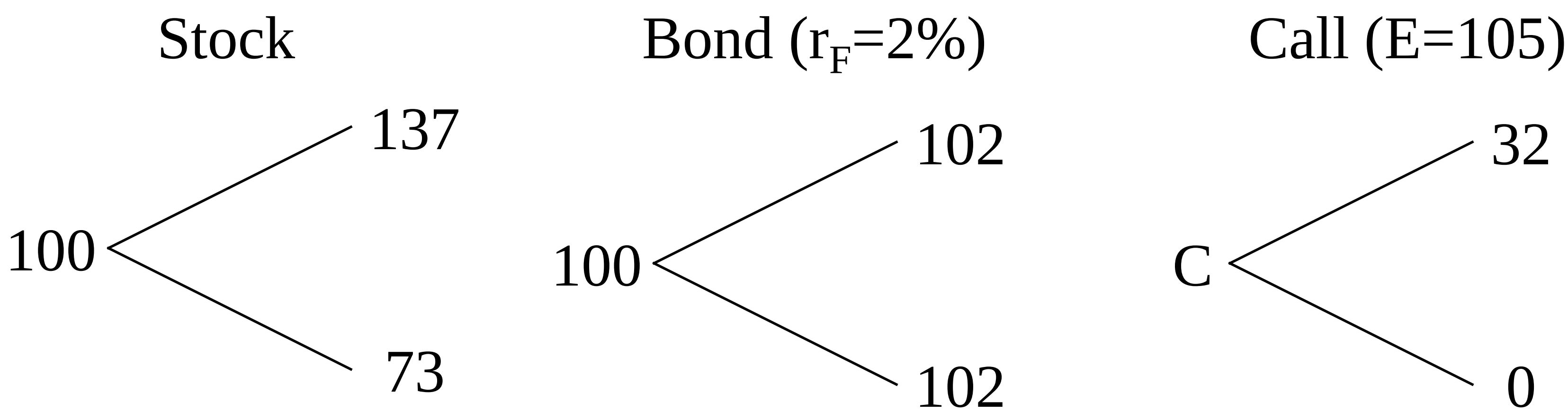
# OptionPricer08.cpp (continue)

```
American put option price = 11.72
American put price tree:
    11.724
    16.734     6.517
    23.161    10.075     2.799
    30.930    15.143     4.788     0.712
    39.657    21.978     8.030     1.388     0.000
    47.585    30.530    13.113     2.704     0.000     0.000
    54.471    39.657    20.611     5.271     0.000     0.000     0.000
    60.453    47.585    30.530    10.273     0.000     0.000     0.000     0.000
    65.649    54.471    39.657    20.023     0.000     0.000     0.000     0.000     0.000
American put exercise policy:
       0
       0       0
       0       0       0
       0       0       0       0
       1       0       0       0       0
       1       1       0       0       0       0
       1       1       0       0       0       0       0
       1       1       1       0       0       0       0       0
       1       1       1       1       1       1       1       1       1
*/
```

## Notes:

- ***BinLattice<double> PriceTree*** and ***BinLattice<bool> StoppingTree***
  - Create objects ***PriceTree*** and ***StoppingTree.***
- ***OptionCalculation.PriceBySnell(Model, PriceTree, StoppingTree)***
  - Compute the option prices and stopping policy for all nodes and store them inside PriceTree and StoppingTree.
- ***PriceTree.Display()*** displays the prices for all nodes.
- ***StoppingTree.Display()*** displays the stopping policy:
  - 1s for the nodes where the American option should be exercised (unless exercised already), and 0s for the others.

# Payoffs




1-year call option, S=100, E=105, $r_F$=2% (annual)
1 step per year
Can the call option payoffs be replicated?

# Replicating Strategy

Buy ½ share of stock, borrow \$35.78 (at the risk-free rate).

Payoff

(½)137 - (1.02) 35.78 = 32

Cost

(1/2)100 - 35.78 = 14.22

Payoff

(½)73 - (1.02) 35.78 = 0

The value of the option is \$14.22!

# Solving for the Replicating Strategy

The call option is equivalent to a levered position in the stock (i.e., a position in the stock financed by borrowing).

$$137\ H - 1.02\ B = 32$$

$$73\ H - 1.02\ B = 0$$

$\Rightarrow$ H (delta) = ½ = $(C^+ - C^-)/(S^+ - S^-)$

$B = (S^+ H - C^+)/(1 + r_F) = 35.78$

Note: the value is (apparently) independent of probabilities and preferences!

## Assignment #1

- Modify the ***PriceByCRR()*** function in ***Options08.h*** and ***Options08.cpp*** to compute the replicating strategy for a European option in the binomial tree model, using the ***BinLattice<>*** class template to store the stock and money market account positions in the replicating strategy at the nodes of the binomial tree. X(n, i) is the delta (fraction) of share of stock at each node on the stock tree, y(n, i) is the money market position at each node on the money market tree. Negative value mean borrowed money:

$$x(n,i) = \frac{H(n+1,i+1) - H(n+1,i)}{S(n+1,i+1) - S(n+1,i)},$$

$$y(n,i) = \frac{H(n+1,i) - x(n,i)S(n+1,i)}{(R)^\wedge(1+n)},$$

for n = 0, 1, …, N-1 and i = 0, 1, …, n, where S(n, i) and H(n, i) denote the stock and option prices at time *n* and node *i*.

```
#include "BinomialTreeModel02.h"
#include "Option08.h"
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
using namespace fre;
int main()
{
   int N = 8;
   double U = 1.15125, D = 0.86862, R = 1.00545;
   double S0 = 106.00, K = 100.00;
   BinomialTreeModel Model(S0, U, D, R);
   ofstream fout;
   fout.open("Results.txt");
   Call call(N, K);
   OptionCalculation callCalculation(&call);
```

```
BinLattice<double> PriceTree;
BinLattice<double> XTree;
BinLattice<double> YTree;

fout << "European call prices by PriceByCRR:"
   << fixed << setprecision(3) << callCalculation.PriceByCRR(Model) << endl << endl;

fout << "European call prices by HW6 PriceByCRR:"
   << fixed << setprecision(3) << callCalculation.PriceByCRR(Model, PriceTree, XTree, YTree)
   << endl << endl;

fout << "Stock positions in replicating strategy:“ << endl << endl;
XTree.Display(fout);

fout << "Money market account positions in replicating strategy:“  << endl << endl;
YTree.Display(fout);

……….
```

# Overloaded Display() member function for BinLattice

```
void Display(ofstream& foutput)
    {
       foutput << setiosflags(ios::fixed) << setprecision(3);
       for (int n = 0; n <= N; n++)
       {
          for (int i = 0; i <= n; i++)
             foutput << setw(15) << GetNode(n, i);
          foutput << endl;
       }
       foutput << endl;
    }
```

## Assignment #2

- The binomial model can be employed to approximate the Black-Scholes model. One of several possible approximation schemes is the following. Divide the time interval [0, T] into ***N*** steps of length ***h = T/N***, and set the parameters of the binomial model to be

$$U = e^{\sigma\sqrt{h}},$$

$$D = \frac{1}{U},$$

$$R = e^{rh},$$

where σ is the volatility and *r* is the continuously compounded interest rate in the Black-Scholes model.

- Develop code to compute the approximate price for an American call option in the Black-Scholes model by means of this binomial tree approximation.

# References


- Numerical Methods in Finance with C++ (Mastering Mathematical Finance), by Maciej J. Capinski and Tomasz Zastawniak, Cambridge University Press, 2012, ISBN-10: 0521177162
- *Financial Instrument Pricing Using C++, Daniel J. Duffy, ISBN 0470855096, Wiley, 2004.*
- *C++ STL, www.cs.mtsu.edu/~jhankins/files/3110/presentations/stl.ppt*
- *Inheritance in C++, cs.njit.edu/maura/ClassNotes/CIS601/lecture6.ppt*
- *Function Templates; C++ Class Templates, web.cse.ohio-state.edu/~neelam/courses/45922/Au05Somasund/PPT/Lecture7.ppt*