Roxana Dumitrescu

C++ in Financial Mathematics

What have we learnt?

- **Inheritance** Create a base class and derived classes, where **the base class contains the common member variables and functions.**
- **Polymorphism**- *Many forms***.** You want a function of the base class to behave in a specific way to each of the derived classes**.** In order to get the polymorphic behaviour: use the keyword **virtual** and **pointers or references** to the base class**.**

What have we learnt?

We have seen the example of a **Musician hierarchy**.

1. Base class

```
class Musician{
public:
virtual void greet();
};

void Musician::greet()  {cout<<"musician says:
   hello\n":}
```

What have we learnt?

2. A first derived class: Trumpeter

```
class Trumpeter: public Musician{
public:
virtual void greet();
};

void Trumpeter::greet()  {cout<<"trumpeter
   says:hello\n":}
```

What have we learnt?

3. A second derived class: Pianist

```
class Pianist: public Musician{
public:
 virtual void greet();
};
void Pianist::greet()  {cout<<"pianist says:
   hello\n":}
```

What have we learnt?

Example of polymorphic code only using pointers to the base class Musician.

```
void Musiciangreet(Musician* pm)
{
cout<<"introducing....\n";
pm->greet();
}
```

This routine will work on an instance of Musician or any class derived from Musician because **greet() is virtual**!

What have we learnt?

We can create an entire orchestra in a single data structure!

```
int main(){
vector<Musician*> orchestra; // a vector which
   holds the entire orchestra
orchestra.push_back(new Trumpeter);
orchestra.push_back(new Pianist);
orchestra.push_back(new Violonist);
for (int i=0; i<orchestra.size(); i++)
Musiciangreet(orchestra[i]);
}
```

Polymorphism

Abstract base classes and pure virtual functions

Often in a design, you want the base class to present *only* an interface for its derived classes. In other words, you don't want anyone to actually create an object of the base class, only to upcast to it, so that its interface can be used. This is accomplished by making that class *abstract* which happens if you give it at least one *pure virtual function*.

What is a pure virtual function?

A pure virtual function is a function which uses the **virtual** keyword and is followed by **=0**.

You can not create an instance of an abstract class. If anyone tries to make an object of an abstract class, the compiler prevents them. This is a tool that allows you to enforce a particular design.

Polymorphism

Abstract base classes and pure virtual functions

When an abstract class is inherited, all pure virtual functions must be implemented.

An interface class (or a pure abstract class) is a class which contains **only pure virtual functions** and **no member variables.** It can be seen as a contract between the designer of the class and the users, in the sense that any class implementing the interface class provides the functionality announced in the interface class.

Polymorphism

The constructor can not be made virtual!

Destructors and Virtual destructors What happens if you want to manipulate an object through a pointer to its base class (that is, manipulate the object through its generic interface)? The problem occurs when you want to **delete** a pointer of this type for an object that has been created on the heap with **new.** If the pointer is to the base class, the compiler can only know to call the base-class version of the destructor during **delete.** This is the same problem that virtual functions were created to solve the general case.

Virtual functions work for destructors as they do for all other functions except constructors.

Polymorphism

Virtual versus non-virtual destructor!

```
//Behavior of virtual vs. non-virtual
   destructor
class Base1{
public:
  ~Base1(){ cout<<"~Base1()\n ";}
};

class Derived1: public Base1{
public: ~Derived1(){cout<<"~Derived1()\n";}

};
```

Polymorphism

Virtual versus non-virtual destructor!

```
//Behavior of virtual vs. non-virtual
   destructor

class Base2{
public:
 virtual ~Base2(){ cout<<"~Base2()\n ";}
};

class Derived2: public Base2{
public: ~Derived2(){cout<<"~Derived2()\n";}
};
```

Polymorphism

Virtual versus non-virtual destructor!

```
//Behavior of virtual vs. non-virtual
   destructor
int main()
{
Base1 *bp=new Derived 1; // Upcast
delete bp;
Base2 * bp2=new Derived2; // Upcast
delete bp2;
}
```

Polymorphism

Virtual versus non-virtual destructor!

When you run the program you'll see:

- **delete bp** only calls the base-class destructor;
- **delete b2p** calls the derived-class destructor followed by the base class destructor, which is the behavior we desire.

Note that:

- Forgetting to make a destructor **virtual** is an insidious bug because it often doesn't directly affect the behavior of your program, but it can quietly introduce a memory leak.
- Even though the destructor, like the constructor, is an "exceptional" function, it is possible for the destructor to be virtual because the object already knows what type it is.

Plan

We’ll learn how to use C++ in order to write a Binomial pricer.

Binomial pricer

Binomial pricer

Binomial model: Mathematical background

- In the **binomial model** the prices of assets evolve in discrete time step $n = 0, 1, 2, ...$ There is a **stock** whose price evolves randomly by moving up a factor $1 + U$ or down by $1 + D$ independently at each time step, starting from the spot price S_0. The stock price becomes:

  $$S(n, i) = S_0(1 + U)^i(1 + D)^{n-i}$$

  at step n and node i in the binomial tree.

We consider $S_0 > 0$ and $U > D > -1$ and $n \geq i \geq 0$.

Binomial pricer

Binomial model: Mathematical background

- There is a risk-free security, a **money market account**, growing by a factor $1 + R > 0$ during each time step.
- The model admits no arbitrage whenever $D < R < U$.
- The **price** $H(n, i)$ at each time step n and node i of a **European option** with expiry date N and payoff $h(S(N, i))$ can be computed using the **Cox-Ross-Rubinstein (CRR)** method.

Binomial pricer

Binomial model: Mathematical background

Cox-Ross-Rubinstein (CRR) procedure: backward induction

- At the expiry date N

  $$H(N, i) = h(S(N, i))$$

  for each node $i = 0, 1, ..., N$
- Fix $n = 0, 1, ..., N - 1$. If $H(n + 1, i)$ is already known at each node $i = 0, 1, ..., n + 1$ then

  $$H(n, i) = \frac{qH(n + 1, i + 1) + (1 - q)H(n + 1, i)}{1 + R} \tag{1}$$

  for each $i = 0, 1, .., n$.

Binomial pricer

Binomial model: Mathematical background

Cox-Ross-Rubinstein (CRR) procedure: backward induction

In the above formula, we have denoted by q the **risk-neutral probability.** It is defined as follows:

$$q = \frac{R - D}{U - D}.$$

Binomial pricer

Binomial model: Mathematical background

Cox-Ross-Rubinstein (CRR) procedure: backward induction
Examples:

- For a **call option** the payoff function is

$$h^{\text{call}}(z) = (z - K)^{+}. \tag{2}$$

- For a **put option** the payoff function is

$$h^{\text{put}}(z) = (K - z)^{+}. \tag{3}$$

K represents the **strike** price.

Binomial pricer

Binomial pricer: C++ design

Implementation: First class

Aim: We want to encapsulate the binomial model consisting of stock and a money market account, while **leaving out** anything related to options.

Solution: In C++ this can be achieved using a class, which will include **the variables** S_0, U, D, R determing the binomial model, and also all **the functions specific to the model**.

Binomial pricer

Binomial pricer: C++ design

Implementation: First class

Functions specific to the model:

- Function which computes the risk-neutral probability

```
double RiskNeutProb() const;
```

- Function which computes the stock price at node n, i

```
double S(n,i) const;
```

- Functions which return R, S_0, U, D

```
double GetR() const;
double GetStock() const;
double GetU() const;
double GetD() const;
```

Binomial pricer

Binomial pricer: C++ design

Implementation: First class

Functions specific to the model:

- Functions which set R, S_0, U, D

```
void SetR() ;
void SetStock();
void SetU();
void SetD();
```

- Function which checks model data

```
void CheckData() const;
```

Binomial pricer

Binomial pricer: C++ design

```
class BinomialModel
{
private:
    double S0;
    double U;
    double D;
    double R;
public:
    double RiskNeutProb() const;
    double S(int n, int i) const;
    double GetR() const;
    double GetStock() const;
    double GetU() const;
    double GetD() const;
```

Binomial pricer

Binomial pricer: C++ design

```
    void SetStock(double);
    void SetU(double);
    void SetD(double);
    void SetR(double);

/* Checking model data*/
    void CheckData() const;
};
```

Binomial pricer

Binomial pricer: C++ design

Remember: the private members S_0, U, D, R are inaccesible outside the class. The public members of the class will be accessible outside the class (we'll see calls to these functions made from other parts of the program).

Binomial pricer

Binomial pricer: C++ design

```
double BinomialModel::RiskNeutProb() const
{
    return (R-D)/(U-D);
}
```

- How is **RiskNeutProb()** going to compute what it needs to compute if we do not seem to be passing anything to it? In fact, it does know U, D, R because it is a member of the same class!

Binomial pricer

Binomial pricer: C++ design

```
double BinomialModel::S(int n, int i) const
{
    return S0*pow(1+U,i)*pow(1+D,n-i);
}
```

- In order to compute the price at the node i, time n, we use the function pow from the cmath library.

Binomial pricer

The member function CheckData of the class BinomialModel

```
void BinomialModel::CheckData() const
{  /* Making sure that 0<S0, -1<D<U, -1<R */
if (S0<=0.0 || U<=-1.0 || D<=-1.0 || U<=D ||
   R<=-1.0)
{  cout<<"Illegal data ranges"<<endl;
    cout<<"Terminating program"<<endl;
    exit(1);
}
/* Checking for arbitrage*/
if (R>=U || R<=D)
    {   cout<<"Arbitrage exists "<<endl;
        cout<<"Terminating program "<<endl;
        exit(1);}
};
```

Binomial pricer

Option hierarchy

Aim: We now create a **second class** representing the option of *European style* (the financial contract). Actually, we'll have a **hierarchy of classes**, in order to be able to price any type of European option!

Binomial Pricer

Option hierarchy

How to identify the hierarchy of classes?

Solution:

- Identification of the common variables and functions;
- Identification of the common functions with polymorphic behaviour;
- Identification of the specific variables and functions to each class.

Binomial pricer

Option hierarchy

We first write **A base class *EurOption*** containing the common elements:

```
class EurOption
{/*Steps to expiry*/
private: int N;
 public:
    void SetN(int);
    int GetN() const {return N;};
```

Binomial pricer

Option hierarchy

```
    /* the payoff*/
    virtual double Payoff(double z) const=0;
     /*Function PriceByCRR which returns the
   price at time 0 */
    double PriceByCRR(const BinomialModel&)
   const;
    virtual ~EurOption(){};
};
```

Binomial pricer

Option hierarchy

Remark:

1. Note that the function payoff is declared **virtual**, because each type of financial contract has its own definition of the payoff!
2. The function **PriceByCRR** computes the price of the European option at time 0.

Binomial pricer

Option hierarchy

```
double EurOption::PriceByCRR(const
   BinomialModel& Model) const
{
    double q= Model.RiskNeutProb();
    vector<double> Price(N+1);
    /*Fix the terminal condition*/
    for (int i=0;i<=N;i++)
    {
        Price[i]=Payoff(Model.S(N,i));
    }
```

Binomial pricer

Option hierarchy

```
     /*For each period n we compute in all the
    nodes*/
     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])
     /(1+Model.GetR());
          }
     }
     return Price[0];
}
```

Binomial pricer

Option hierarchy

We now write a class **Call** and a class **Put**, both **derived from the class *EurOption*** (which should contain the specific implementations of the function **payoff**!):

```
class Call: public EurOption
{
private: double K;
public:
     double getK() const{return K;};
     void setK(double);
    double Payoff(double z) const;
    ~Call(){};
};
```

Binomial pricer

Option hierarchy

```
class Put: public EurOption
{
private: double K;
public:
     double getK() const{return K;};
     void setK(double);
    double Payoff(double z) const;
    ~Put(){};
};
```

Binomial pricer

Option hierarchy

```
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;
}
```

Binomial pricer

American options

- Multiple inheritance
- Virtual inheritance
- Class templates

Binomial pricer

American options: Mathematical background

In addition to pricing **European options**, we want to include the ability to price **American options** in the binomial model.

The holder of an **American option** has the right to exercise it at any time up to and including the expiry date N. If the option is exercised at time step n and node i of the binomial tree, then the holder will receive payoff $h(S(n,i))$.

Binomial pricer

American options: Mathematical background

The price $H(n, i)$ of an American option at any time step n and node i in the binomial tree can be computed by the following procedure, which proceeds by backward induction on n:

- At the expiry date N

$$H(N, i) = h(S(N, i)) \tag{4}$$

  for each node $i = 0, 1, ..., N$.

Binomial pricer

American options: Mathematical background

- If $H(n+1,i)$ is already known at each node $i = 0,1,...,n+1$ for some $n = 0,...,N-1$, then

$$H(n,i) = \max\left(\frac{qH(n+1,i+1)+(1-q)H(n+1,i)}{1+R}, h(S(n,i))\right) \tag{5}$$

  for each node $i = 0,1,...,N$.

In particular, $H(0)$ at the root note of the tree is the price of the American option at time 0. Note that the discounted process $(1+R)^{-n}H(n,i)$ is the **Snell envelope** of the discounted process $(1+R)^{-n}h(S(n,i))$.

Binomial pricer

American options

Multiple inheritance

Just as for European options, we can distinguish subclasses of American options according to the payoff, such as puts, calls, digital calls, bull spreads and many others.

We have seen how to use inheritance in C++ to set up subclasses. A **new feature** is that an option with a particular payoff such as a put can be either of European or American type. This can be modelled in C++ by means of multiple inheritance.

Binomial pricer

American options

Multiple inheritance

- In addition to the **EurOption** class, we define a new **AmOption** class, and make **Put** a subclass of both the **EurOption** and **AmOption** classes.
- Similarly, **Call** (and possibly more classes for other payoffs) will also be inheriting from the **EurOption** and **AmOption** classes.

Binomial pricer

American options

Multiple inheritance

```
class AmOption
{ private: int N;
public:
    /*Function PriceBySnell which returns the
   price at time 0 */
    void SetN(double);
    int GetN() const;
    double PriceBySnell(const BinomialModel&)
   const;
    virtual ~AmOption(){};
};
```

Binomial pricer

American options

Multiple inheritance
We encounter some new features:

- A new class **AmOption** is introduced, similar to the **EurOption** class.
- **PriceBySnell()** in the **AmOption** class replaces the **PriceByCRR()** function from the **EurOption** class.
- Because puts and calls can be either of European and American type, the **Call** and **Put** classes inherit from both the **EurOption** and **AmOption** classes.

Binomial pricer

American options

Multiple inheritance

The line

```
class Put:public EurOption, public AmOption
```

declare **Put** as a **public** subclass of the **EurOption** class as well as of the **AmOption** class. Similar for the **Call** class.

Binomial pricer

American options

Multiple inheritance

```
double AmOption::PriceBySnell (const BinModel&
   Model) const
{
double q=Model.RiskNeutProb();
vector<double>Price(N+1);
double ContVal;
for (int i=0;i<N;i++)
{   Price[i]=Payoff(Model.S(N,i));
}
```

Binomial pricer

American options

Multiple inheritance

```
for (int n=N-1; n>=0;n--)
    {
for (int i=0;i<=n;i++)
{
ContVal=(q*Price[i+1]+(1-q)*Price[i])/(1+Model.GetR(
Price[i]=Payoff(Model.S(n,i));
if (ContVal>Price[i])   Price[i]=ContVal;
}
    }
return Price[0];
}
```

Binomial pricer

American options

Multiple inheritance

- The main new addition is the function **PriceBySnell()** belonging to the **AmOption** class. The Snell envelope procedure is implemented in this function, returning the American option price at time 0.

Binomial pricer

American options

Multiple inheritance

- If we add a function **function_test** in the class Call (or Put) and we want to set the expiry date *N*, then we have to write:

```
EurOption::SetN(N);
AmOption::SetN(N);
```

Binomial pricer

American options

Virtual inheritance

- The code developed above to include American options **suffers from a drawback.** The **EurOption** class and the **AmOption** class each have their own variable N to store the expiry date.
- The variable N had to be initialised by the same value separately for European and American options. It is redundant to have two copies of the same number. The expiry date is a common feature shared by all options.
- We need a single copy of N shared between **EurOption** and **AmOption** classes.
- Similar remarks apply to the function **SetN()** and the **virtual** function **Payoff()**.

Binomial pricer

American options

Virtual inheritance

- It would be much more sensible to have a single copy of each of these functions shared between **EurOption** and **AmOption** classes than two copies owned separately by each of these two classes.

How do we achieve this?

- We create a new class, and call it the **Option** class, to contain the common features shared by all options such as expiry date and a payoff function. Then we change the **EurOption** and **AmOption** classes to become subclasses of the **Option** class, so they will inherit these features.

Binomial pricer

American options

Virtual inheritance

```
class Option
{
private: int N; //steps to expiry
public:
    void SetN(int N){this->N=N;};
    int  GetN() const{return N;};
    virtual double Payoff(double z)=0;
    virtual ~Option(){};
}
```

Binomial pricer

American options

Virtual inheritance

```
class EurOption: public virtual Option
{
public:
    // pricing European option
    double PriceByCRR(const BinModel& Model);
    virtual ~EurOption(){};
}
```

Binomial pricer

American options

Virtual inheritance

```
class AmOption: public virtual Option
{
public:
    // pricing American option
    double PriceBySnell(const BinModel& Model);
    virtual ~AmOption(){};
}
```

Binomial pricer

American options

Virtual inheritance

```
class Call: public EurOption, public AmOption
{
private:
double K;  //strike price
public:
void SetK(double K){this->K=K;};
double GetK() const{return K;};
double Payoff(double z);
~Call(){};
}
```

Binomial pricer

American options

Virtual inheritance

The benefit of virtual inheritance:

```
int Call::function_test()
{       int N=10;
        SetN(N);
}
```

Binomial pricer

American options

Virtual inheritance

We analyse the changes in the previous code:

- A new class **Option** is introduced. The variable *N*, **GetN()**, **SetN()** and virtual function **Payoff()** are moved into this class.
- The line setting *N* inside **function_test()** has become

  ```
  SetN(N);
  ```

  No longer will there be a need to set *N* separately for the **EurOption** and **AmOption** classes!

Binomial pricer

American options

Virtual inheritance

- **EurOption** is declared a subclass of the **Option** class by

```
class EurOption:public virtual Option
```

- There is a similar line for the **AmOption** class. These classes no longer explicitly contain *N*, **SetN()**, **GetN()** or **Payoff()**, but inherit these members from the **Option** class.

Binomial pricer

American options

There is a new feature here that must be explained: the role of the keyword **virtual** in this context. This brings us to the topic of **virtual inheritance**.

- Let us temporarily remove the keyword **virtual**, so the above line becomes

  ```
  class EurOption: public Option
  ```

  with a similar change in the corresponding line for the **AmOption** class.
- **Problem** When the file is compiled, an **error message** will be produced. Depending on the compiler, it may read something like this:

  "reference to Set(N) is ambigous".

Binomial pricer

American options

Virtual inheritance

- The reason for the error message is this. Without virtual inheritance (i.e. with the keyword **virtual** removed) the **EurOption** and **AmOption** classes inherit their own distinct copies of N from the **Option** class.

  In turn, the **Call** class inherits two distinct copies of N, one via the **EurOption** class and one via the **AmOption** class. When the function **Call::function_test()** tries to execute **SetN(N)**, it does not know which of these two copies of N is referred to.

Binomial pricer

American options

Virtual inheritance

- When virtual inheritance applies (that is, the keyword **virtual** is reinstated), the code compiles and runs without a complaint. In this case a single copy of N (and indeed of any other member of the **Option** class) is shared between the subclasses **EurOption** and **AmOption**. It is this single shared copy of N that is then inherited by the **Call** class.

Binomial pricer

American options

Remark concerning multiple inheritance: Our example studied above (the class call which is derived from both EurOption and AmOption) shows that multiple inheritance has to be used carefully (in our case, we had to use virtual inheritance). It is preferable to use multiple inheritance only with interface classes.

Binomial pricer

Templates

Binomial pricer

Class templates

- **Aim** We would like to compute and to 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 steps n and noded i at which the option should be exercised are characterised by **the condition**

$$H(n,i) = h(S(n,i)) > 0.$$

Binomial pricer

Class templates

- The idea will be to encode this information as data of type **bool**, taking just two possible values, 0 is the option should not be exercised at a given node, or 1 otherwise, depending on whether the above condition is violated or not.
- The natural structure for the price data is that of a lattice indexed by time steps $n = 0, 1, ..., N$ and nodes $i = 0, 1, ..., n$.

Binomial pricer

Class templates

- The convenient way to store the option prices will be a vector indexed by time variable *n* consisting vectors of type **double** indexed by the nodes *i* at each time *n*. We'll thus have a vector of vectors whose declaration is the following

```
vector<vector<double>> Lattice;
```

  We also provide related functionality such as setting the number of time steps *N* or setting and retrieving a value at time *n* and node *i*.

Binomial pricer

Class templates

```
class BinLattice
{
private:
   int N;
   vector<vector<double>> Lattice;
        public:
   void SetN(int N)
{
this->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;}
```

Binomial pricer

Class templates

```
   double GetNode(int n, int i) const
{return Lattice[n][i];}
   void Display() const
{
for (int n=0;n<=N;n++)
{
for (int i=0;i<=n;i++)
cout<<GetNode(n,i)<<" ";
}
cout<<endl;
}
}
```

Binomial pricer

Class templates

The **BinLattice** class is defined here.

The class contains two variables:

- N to store the number of time steps in the binomial tree
- **Lattice**, a vector of vectors to hold data of type double

The **BinLattice** class also contains the following definitions:

- The **SetN()** function takes a parameter of type **int**, assigns it to N, sets the size of the **Lattice** vector to $N+1$, the number of time instants n from to 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.

Binomial pricer

Class templates

- **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 values stored at step *n*, node *i*.

Binomial pricer

Class templates

How to record the stopping policy?

We remark that in order to record the stopping strategy we need exactly the same class BinLattice, but with data of type **bool** instead of **double**. However, we do not want several duplicate code!

Solution?

Class templates offer a much neater solution. The type is not hardwired inside the class, but passed to it as a parameter. We achieve this by modifying the **BinLattice** class as follows.

Binomial pricer

Class templates

How to record the stopping policy?

```
template<typename Type> class BinLattice
{ private:
   int N;
   vector<vector<Type>> Lattice;
        public:
   void SetN(int N)
{      this->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;}
```

Binomial pricer

Class templates

```
   Type GetNode(int n, int i) const {return
   Lattice[n][i];}
   void Display() const
{    for (int n=0;n<=N;n++)
{
for (int i=0;i<=n;i++)
cout<<GetNode(n,i)<<" ";
}
cout<<endl;
}
}
```

Binomial pricer

Which are the changes?

- **class BinLattice** is replaced by **template<typename Type> class BinLattice** which specifies that **BinLattice** is no longer a class, but a **class template** with **type parameter** Type. **Every occurence of double is replaced by Type**
- **vector<vector<double>> Lattice;** is replaced by **vector<vector<Type>> Lattice;**
- **void SetNode(int n, int i, double x)** is replaced by **void SetNode(int n, int i, Type x);**
- **double GetNode(int n, int i)** is replaced by **Type GetNode(int n, int i);**

Binomial pricer

Which are the changes?

Remark: The previous code (declaration and definition of the template class BinLattice) has to be written in a header (.h) file. There is no .cpp file corresponding to BinLattice. 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, and we have not done so yet. **Separate compilation will not work** for them.

Binomial pricer

How should we modify the PriceBySnell function?

```
double PriceBySnell(const BinModel& Model,
   BinLattice<double>& PriceTree,
   BinLattice<bool>& StoppingTree);
```

Binomial pricer

How should we modify the PriceBySnell function?

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

Binomial pricer

How should we modify the PriceBySnell function?

```
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)/(1+Model.GetR());
PriceTree.SetNode(n,i,Payoff(Model.S(n,i)));
StoppingTree.SetNode(n,i,1);
```

Binomial pricer

How should we modify the PriceBySnell function?

```
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);
}
```

Summing up

- We have written a **Binomial pricer** which contains:
  - A class representing the model (the evolution of the asset price): **class BinomialModel**
  - An hierarchy of classes representing the financial contract (of *European and American type*)
  - A class **BinLattice** which is used in order to store the option price at each node *i* and time step *n*, as well as the stopping strategy in the case of American options
- C++ Design Techniques: inheritance (also multiple inheritance and virtual inheritance), polymoprhism, some elements of generic programming (templates)**.**