FINM 326: Computing for Finance in C++

Lecture 9

Chanaka Liyanarachchi

March 3, 2023

Binomial Tree Pricer

Electronic Trading

Course Wrap-up

Binomial Tree Pricer

Notation

We will use the following notation:

S_t	:	Stock price at time t
σ	:	Volatility of the stock
r	:	Interest rate
T	:	Time to option expiration (in years)
K	:	Strike price
W_t	:	Brownian motion process
$N(0,1)$	:	Standard Normal Distribution
$(A-B)^+$	:	Max(A-B, 0)

Stock Price SDE

- We assume a stock price process follows a Geometric Brownian Motion.
- Under the risk neutral measure the process of the stock price is given by the SDE:

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{1}$$

where,

$$dW_t = N(0, dt) \tag{2}$$

$$W_t = \sqrt{t}N(0,1) \quad \because W_0 = 0 \tag{3}$$

I.e. The Brownian motion has a normal distribution with mean zero (0) and variance t.

- Using the above SDE we get:

$$S_t = S_0 \exp^{((r-\sigma^2/2)t+\sigma\sqrt{t}N(0,1))} \tag{4}$$

The Jarrow-Rudd Tree

The Jarrow-Rudd Binomial Tree

- We will use a binomial tree to simulate the paths of the underlying stochastic process[1].
- Jarrow-Rudd tree assumes the random value (z=N(0, 1)) takes +1 or -1 with equal (risk neutral) probability of 0.5 at every node in the tree.
- If we know the stock price at time t, Equation 4 gives us possible values for the stock prices at time $t + \Delta t$.

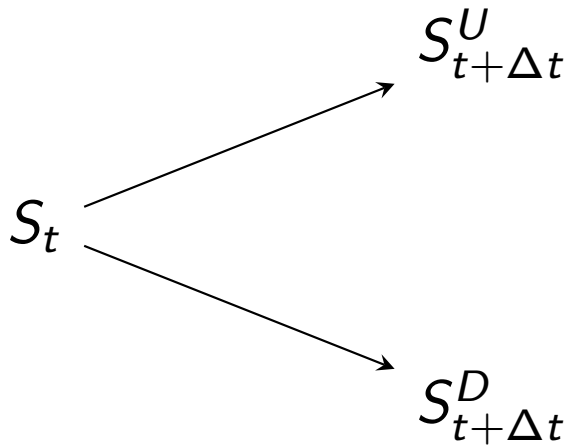
[1] https://en.wikipedia.org/wiki/Geometric_Brownian_motion

Stock Price Process

- This leads to stock price at time $t + \Delta t$:

$$z = 1 : S^U_{t+\Delta t} = S_t \exp^{((r-\sigma^2/2)\Delta t + \sigma\sqrt{\Delta t})} \tag{5}$$

$$z = -1 : S^D_{t+\Delta t} = S_t \exp^{((r-\sigma^2/2)\Delta t - \sigma\sqrt{\Delta t})} \tag{6}$$



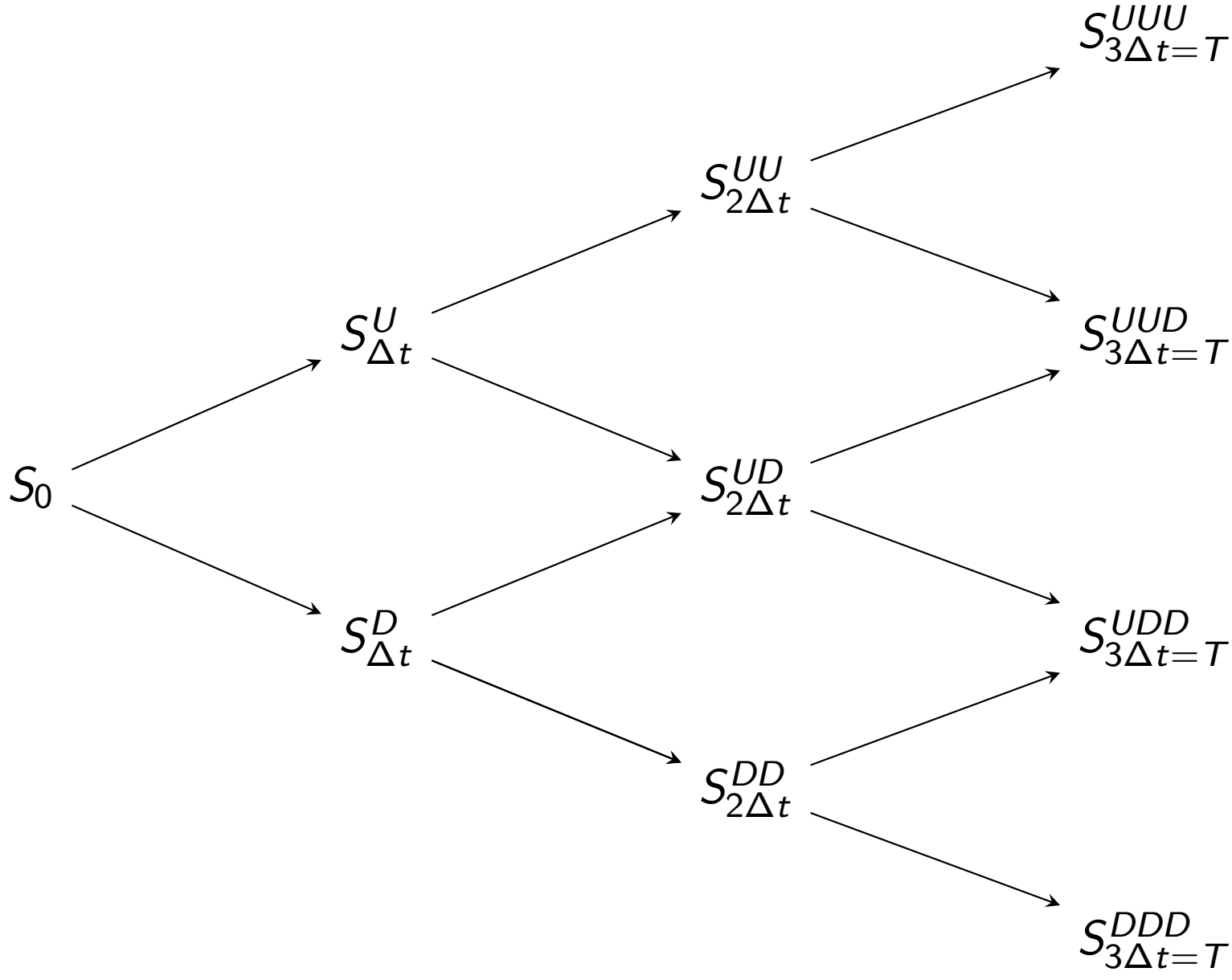
- We refer to these two stock prices as $S^U_{t+\Delta t}$ and $S^D_{t+\Delta t}$ to mean the stock prices at *up* and *down* nodes respectively, at time $t + \Delta t$.

- We simulate how the stock prices would evolve on the tree using Equations 5 and 6:
  - divide time to expiration, T, into equal N parts such that $\Delta t = T/N$
  - discretize W_t such that

$$\Delta W_t = \sqrt{\Delta t} N(0,1) \tag{7}$$

    and assume the underlying Brownian motion can only move a fixed amount up or down at each time step.
  - start from left of the tree, i.e. $t = 0$ and build the binary tree for the stock price process at every Δt increment

- A full binomial tree with 3 times steps (N=3) is shown next.



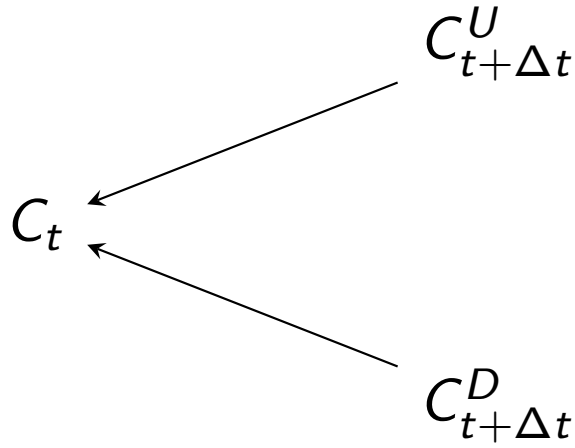
- This construction gives us a binomial tree.
- This tree recombines by construction.
- Recombining means: if you start from a node at time t, an *up* move at time t followed by a *down* move at time $t + \Delta t$ and a *down* move at time t followed by an *up* move at time $t + \Delta t$ end up at the same node at time $t + 2\Delta t$
  $S^{D}_{\Delta t} = S_0 \exp^{((r-\sigma^2/2)\Delta t - \sigma\sqrt{\Delta t})}$
  $S^{DU}_{\Delta t} = S^{D}_{\Delta t} \exp^{((r-\sigma^2/2)\Delta t + \sigma\sqrt{\Delta t})} = S_0 \exp^{2(r-\sigma^2/2)\Delta t} = S^{UD}_{\Delta t}$

- Recombining trees do not grow very large when we increase the number of time steps.
- Note:
  1. The arrows show the direction of *path simulation*.
  2. Superscripts D, DD, U, UU etc. denote how many levels the stock price has gone up or down from the intial value.
     U denotes an up movement; D denotes a down movement.

Pricing European Options

- European options are exercised at maturity (at time T).
- Option payoff at time t=T, i.e. C_T, is given by the payoff function.
  - Call: $C_T = (S_T - K)^+$
  - Put: $C_T = (K - S_T)^+$
- We know S_T at every node; we can find C_T at every node.
- How do we find C_0?

- The option payoff at time t, C_t, is given by the discounted expected value of the option payoffs:

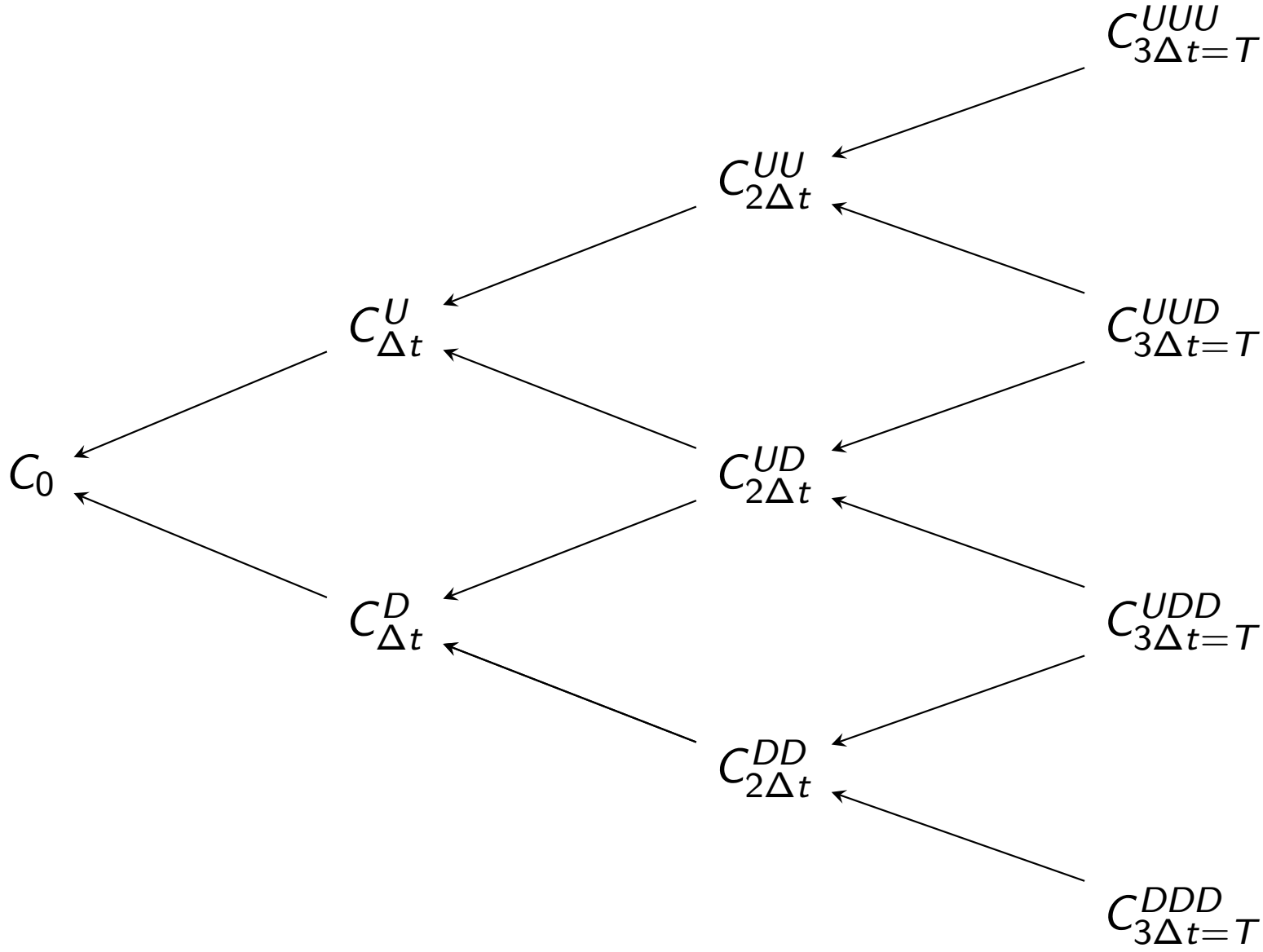


$$C_t = [p_u C^U_{t+\Delta t} + p_d C^D_{t+\Delta t}] exp^{-r\Delta t} \qquad p_u = p_d = 0.5 \quad (8)$$

- Risk neutral probabilities associated with each path (p_u and p_d) is 0.5 (by construction) for Jarrow-Rudd trees.

- We start from the right ($t = T$); back-propagate the payoffs to find C_0.
  Notes:
  1. The arrows show the direction of the calculation.
  2. Superscripts D, DD, U, UU etc. correspond to the stock price levels from the initial value.
  3. 0.5 Risk neutral probability value in Equation 9 applies to Jarrow-Rudd trees only. Other types of trees may have different risk neutral probabilities.

- A full binomial tree with 3 time steps (N=3) is shown below:



- Estimate of the option price at time zero ($t = 0$) is given by C_0.

Pricing Path Dependent Options

- Trees are rarely used in practice to price European style options:
  - price depends on S_T only
  - analytical solutions (e.g. Black Scholes formula) are computationally more efficient
- Trees provide a convenient way to price path dependent options for which analytical solutions are not readily available.
- Path dependent options that can be priced on a tree include:
  1. American
  2. Barrier (`http://en.wikipedia.org/wiki/Barrier_option`).

Pricing American Options

- Allows early exercise.
- We have to consider two payoffs:
  1. Discounted expected payoff from continuing to hold the option:
     - The option payoff at time t, C_t, is given by the discounted expected payoffs:

$$C_t = [p_u C^U_{t+\Delta t} + p_d C^D_{t+\Delta t}] exp^{-r\Delta t} \qquad p_u = p_d = 0.5 \quad (9)$$

  2. Payoff from early exercise
     - Call : $C_t = (S_t - K)^+$
     - Put : $C_t = (K - S_t)^+$

- An investor will use the more profitable choice.
- At each node we need to decide if we exercise the option (early) or hold it:
  - if early exercise is more profitable: use the early exercise payoff
  - otherwise, used discounted expected payoffs

Pricing Barrier Options

- We need to evaluate if the barrier is hit.
- Otherwise, the calculation is similar to their European or American counterparts.

Pricing an Option: Steps

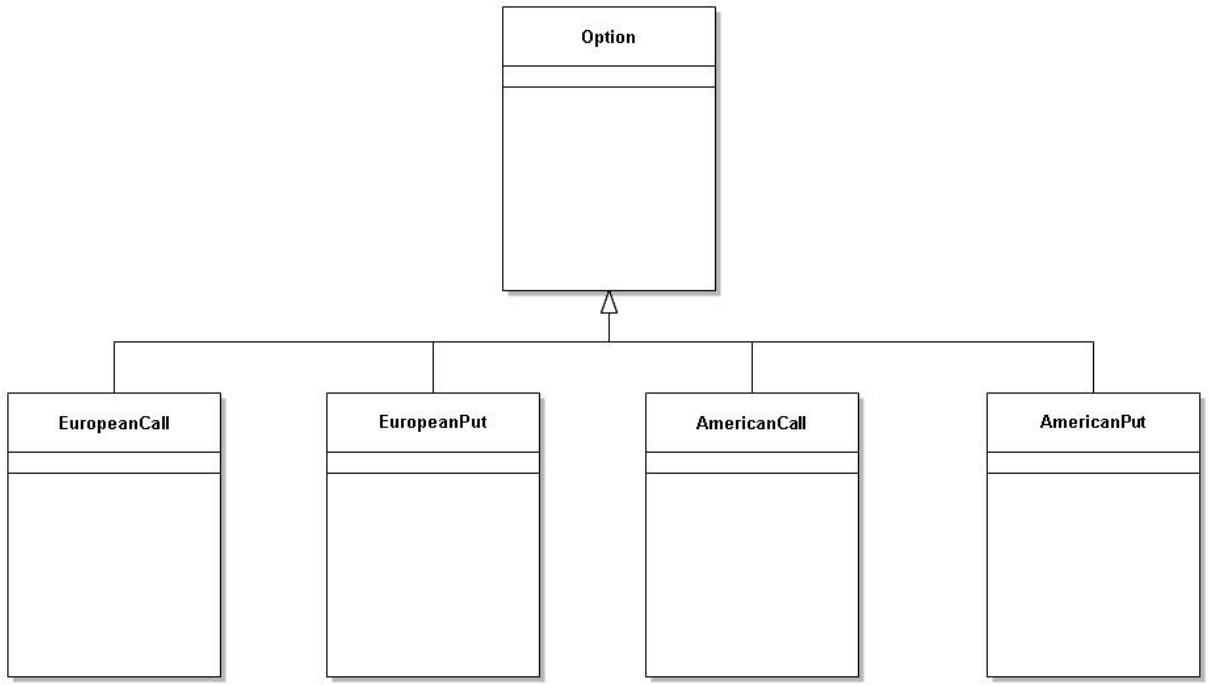
Calculating the option price on a tree involves:

1. Create/initialize the tree using the *tree* described above.
2. Populate the tree with the stock prices from left to right.
3. Calculate the option payoffs on the right-most nodes ($t = T$), using the payoff function of the option.
   E.g. For *regular* European/American:
   - Call: $(S_T - K)^+$
   - Put: $(K - S_T)^+$
4. Starting from the right, back propagate the option payoffs to time zero:
   - European: use discounted expected payoffs
   - American: use discounted expected payoffs and early exercise
   - Barrier: pay attention to barrier levels and rules (in/out)

C++ Implementation

Class Design

- We want to handle different option types (Put, Call) and different styles (European, American, ...).
- One simple design to support code reuse and extensibility:



- Other class designs/hierarchies are possible.

- How do we want to price an option using a tree?
- Last week wrote a MCPricer:

```
int main()
{
   MCPricer pricer(/* args */) ;

   EuropeanCall call(K, T);

   double callPrice = pricer.Price(call);

   cout << "Call Price: " << callPrice << endl;
}
```

- Wouldn't it be convenient if we can just replace the MCPricer with a Tree pricer to use a tree?

Option Base Class

- The `Option` is an abstract base class.
- Option base class already supports:
  - `GetExpirationPayoff()`: gives the option value at option maturity ($t = T$)
    - depends on stock price (S_T)
- We need another function to find the payoff at intermediate nodes:
  - `GetIntermediatePayoff()`: gives the option value at any intermediate node
  - the intermediate payoff may depend on:
    1. all options: the discounted expected payoff
    2. path dependent options: stock price

Option Base Class (Incomplete)

```
class Option
{
public:

   virtual double GetExpirationPayoff(double ST) const = 0;

   virtual double GetIntermediatePayoff(double St,
                  double discountedExpectedPayoff) const = 0;

   ...
   ...
};
```

Derived Option Classes

- Option derived classes (EuropeanCall, AmericanCall etc.) need to implement the 2 pure virtual functions.
- `GetExpirationPayoff()` returns payoff at option maturity (time=T)
- `GetIntermediatePayoff()` returns payoff at any intermediate steps:
  - European options: same as discounted expected value.
  - American options: larger of discounted expected and early exercise.
  - Barrier options: apply barrier rules.

BinomialTree Class

- The `JRBinomialTree` class implements a Jarrow-Rudd binomial tree.
- It supports:
  - `Price()` – takes a reference to an `Option` base class object and prices the given option using the tree.

    ```
    double Price(const Option& option);
    ```

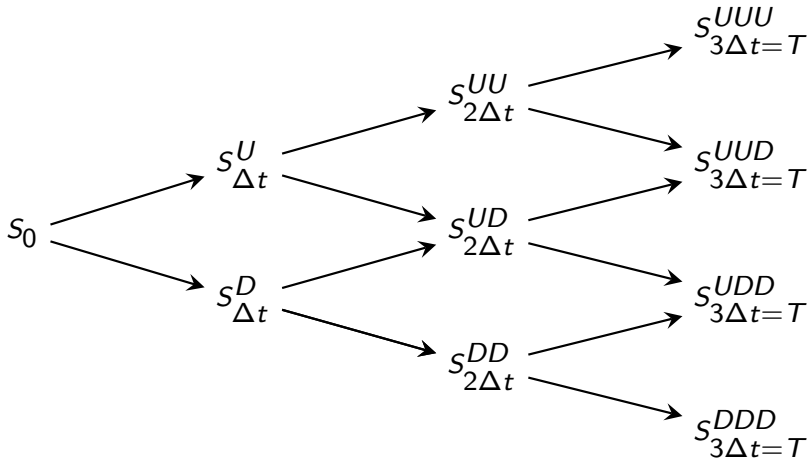
  - Derived (Option) classes are polymorphic.
  - We can pass an object of any derived (Option) class using a reference to a base (Option) class object.

JRBinomialTree Class (incomplete)

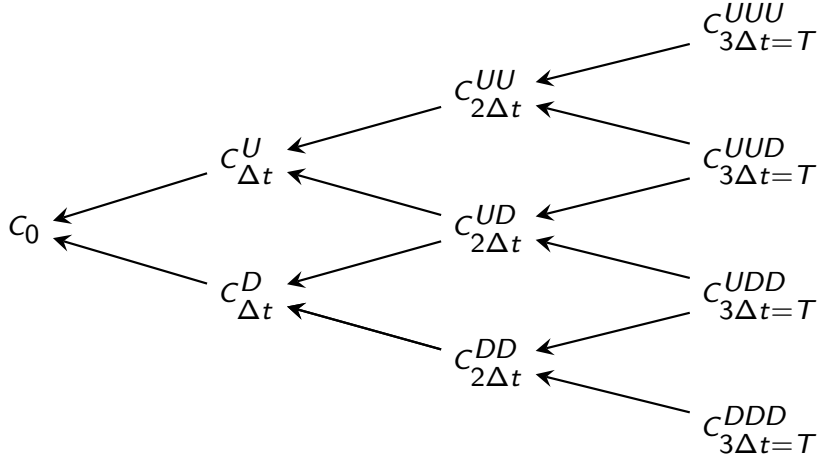
- Class below (incomplete) shows a tree:

```
class JRBinomialTree
{
public:

   double Price(const Option& option);

   ...
   ...
};
```

- How do we represent a binomial tree in a program?
- We have a stock price tree:



- And, an option price tree:



Representing a Node

- Each node has two prices: stock price, option price.
- We have several options to store two prices:
- Using a struct:

```
struct Node
{
   double stockPrice ;
   double optionPayoff;
};
```

- Using a class:

```
class Node
{
public:
   double stockPrice;
   double optionPayoff;
};
```

- We can use a `std::pair` to hold the values at a node:

```
using Node = std::pair<double, double>;
```

Representing a Tree

- A tree has a 2-D structure – height grows when we go from left to right.
- Shown below is a construction[2] of a tree using std::vector that allow us to access each node using two indexes (e.g. `tree[i][j]`):
  - We will use a `vector<Node>` to store values at a given time index, t:

    ```
    using NodesAtTimeStep = vector<Node>;
    ```

  - We will have $N + 1$ `NodesAtTimeStep` vectors.
  - We use another vector to store all `NodesAtTimeStep` vectors; that's our Tree:

    ```
    using Tree = vector<NodesAtTimeStep>;
    ```

    This is the same as:

    ```
    using Tree = vector<vector<Node>>;
    ```

[2] other constructions are possible

std::vector (Detour)

std::vector

By default, use vector when you need a container
– Bjarne Stroustup

- The `vector` is a widely used general purpose container.
- You can think of it as a *smart (C-style) array* with:
  - fast random access
  - ability to grow (resize) dynamically on demand
  - richer functionality
- Defined in `<vector>`.
- `http://www.cplusplus.com/reference/vector/vector/`

Vector: "Size" Related Attributes

- std::vector has two important attributes:
  - capacity: number of elements it currently has space for
  - size: number of elements it currently stores
- Vectors support following *size* related operations:
  - `size()`: returns the number of elements currently stored
  - `capacity()`: returns the number of elements it has space for
  - `empty()`: checks if it is empty (i.e. size is zero)

Vector: Construction

std::vector supports several constructors:

1. Default constructor creates an empty vector:

   ```
   std::vector<int> v;
   ```

   - v has capacity=0; size=0

2. Constructing an vector with N elements:

   ```
   std::vector<int> v(10);
   ```

   - v has capacity=10; size=10
   - 10 items are default constructed and stored in v.

Reserve

- We can reserve capacity without default constructing elements:

```
std::vector<int> v;
v.reserve(10);
```

- v has capacity=10; size=0

Resize

- The std::vector supports `resize()` which changes the size of the container:

```
std::vector<int> v;
v.resize(10);
```

- v has size=10 and capacity=10
- Elements are default constructed.

std::vector: Example 1

- Code below shows how to insert values to a vector of integers:

```
#include <vector>

using std::vector;

int main()
{
  vector<int> v;

  for (int i=0; i<10; ++i)
  {
     v.push_back(i);
  }
}
```

- Using the default constructor (capacity $= 0$).
- Using `push_back()` to add an element to the back of the vector.
- Push back adds capacity when space is not available. Creating capacity this way is an expensive operation. Why?

std::vector: Example 2

- Code below shows how to access elements in a vector:

```
#include <vector>

using std::vector;

int main()
{
  vector<int> v(10);

  for (int i=0; i<10; ++i)
  {
      v[i] = i;
  }
}
```

- Creating a vector with capacity $= 10$.
- Accessing elements using an index.

std::vector: Example 3

- Code below shows how to read (access) elements using an index:

```
#include <vector>

using std::vector;

int main()
{
  vector<int> v;

  for (int i=0; i<10; ++i)
  {
      v.push_back(i);
  }

  for (int i=0; i<v.size(); ++i)
  {
      cout << v[i] << endl;
  }
}
```

std::vector: Example 4

- Code below shows how to read (access) elements using an iterator:

```
#include <vector>

using std::vector;

int main()
{
  vector<int> v;

  for (int i=0; i<10; ++i)
  {
      v.push_back(i);
  }

  for (auto iter = v.begin(); iter != v.end(); ++iter)
  {
      cout << *iter << endl;
  }
}
```

JRBinomialTree Class (incomplete)

```
struct Node
{
   double S;
   double C;
};

class JRBinomialTree
{
public:
   JRBinomialTree(double S0, double rate,
                  double vol, double expiry,
                  int steps);

   double Price(const Option& option);

private:
   using Tree = vector<vector<Node>>;

   Tree tree_;
};
```

- Note how we pass a reference to the base `Option` class and use it to get the actual payoff using the dynamic type of the object.

```
int main()
{
   BinomialTree tree(S0, r, v, T, N);

   EuropeanCall call(K, T);

   double callPrice = tree.Price(call);

   cout << "Call Price: " << callPrice << endl;

   AmericanPut put(K, T);

   double putPrice = tree.Price(put);

   cout << "Put Price: " << putPrice << endl;
}
```

Assignment 6 (Optional)

1. Implement Jarrow-Rudd Tree Pricer. You may complete the implementation provided, or write your own classes. Any case, you should be able to use a main program similar to the one shown above.
2. Use it to price the following options:
   - 2.1 European Call and Put
   - 2.2 American Call and Put

   $K = 100.0$
   $S_o = 100.0$
   $r = 0.05$
   $\sigma = 0.3$
   $T = 1$ (year)
   Number of time steps $= 1000$
3. Write the results to Console.
4. Due: March 10 by midnight (CST)
5. Everyone is strongly encouraged to complete and submit this assignment. Due to shortened quarter, this assignment will not contribute to your final grade.

Extensible Option Pricer

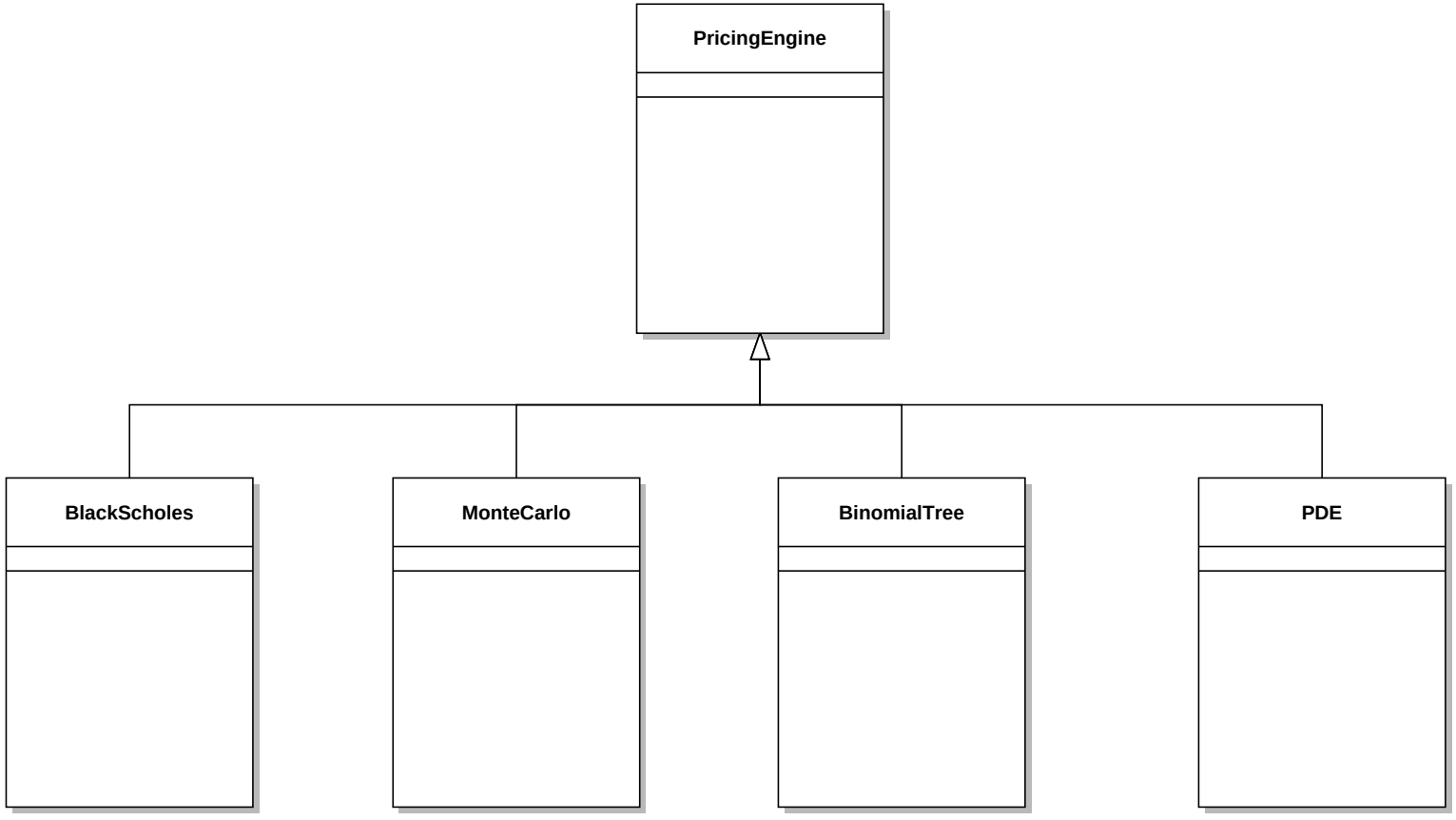
More Pricers

- We wrote several option pricers (Analytical, Monte Carlo, Tree).
- You can follow the binomial tree design to write trinomial tree pricer.
- Trinomial tree and explicit finite difference pde solver are very similar.
- Using what you have to create a explicit finite difference pde is pretty straight-forward.
- We want to add new pricers in an extensible way.

Pricing Engine

- Let's introduce an abstract PricingEngine class:

```
class PricingEngine
{
    virtual double Price(const Option& ) const = 0;
};

class MCPricer : public PricingEngine
{
    double Price(const Option& ) const override;
};

class JRTreePricer : public PricingEngine
{
    double Price(const Option& ) const override;
};
```



- We can use this design to add new engines in an extensible way.
- Homework: Use this design to add new pricers.

Option Pricers: Concluding Remarks

- You will be using option pricers such as the one we developed in class when you join the industry.
- The pricing libraries you see most likely will have more features. Remember, we built this in just 3 weeks.
- You will need to understand them (read code), use them, make changes (bug fixes) and add new features to them.
- I cannot promise that you will understand everything about the libraries based on what we covered in class. What we can do in 9 weeks is very limited.
- I can promise you that the material we covered will go a long way towards your success.
- QuantLib library is a good example of a Quant Finance library you're likely to see in the industry:
  `https://www.quantlib.org/`

Electronic Trading

Electronic Trading: Simplified Overview

- A simplified overview of trading:
  1. Buyers and sellers send orders to the exchange.
  2. The exchange matches the buy orders and the sell orders and we have trades.
  3. The exchange sends information about the trade to the participants of the trade and the rest of the world.
- In electronic trading, steps above are done electronically.

Electronic Trading: Sending an Order

- We need to communicate some of the following information to the exchange:
  - identification, i.e. account info
  - what we want to do: buy/sell, cancel an order etc.
  - product we're going to trade. e.g. TSLA stock
  - quantity to buy/sell. e.g. 100 stocks
  - order type. e.g. market/limit order
  - and, more ...

Electronic Trading - Why Do We Need a Protocol?

- We need to send this information:
  - as **quickly and efficiently** as possible
  - in a way the exchange/other-party understands it **clearly**
- Exchanges support different electronic trading protocols:
  - Proprietary API: developed by the exchange for the use of its client/member firms
  - FIX based: based on the Financial Information eXchange (FIX)

The FIX Protocol

The Financial Information eXchange (FIX) Protocol

- The FIX:
  - is an industry driven **specification**
  - to exchange (send/receive) **trade related information** electronically
  - *managed* by a group of members (`https://www.fixtrading.org`)
- The FIX uses *messages* to transmit trade related information.
- FIX message types:
  - new order
  - cancel
  - cancel and replace
  - execution reports
  - market data
  - and more

QuickFIX

- QuickFIX is an open source FIX implementation written in C++.
- Download at: `http://www.quickfixengine.org/`
- Online documentation available at: `https://quickfixengine.org/c/documentation/`
- QuickFIX uses OOP techniques.
- Today, goal is to:
  - Show how concepts/techniques we discussed in this course are used in QuickFIX
  - And, to show how they can be used to write trading apps, using QuickFIX.
  - We won't discuss how to write a trading app . It takes time to write a trading app.

Message Types

- Messages are implemented using classes.
  - NewOrderSingle class to send a new order.
  - OrderCancelRequest class to cancel an order.
  - OrderCancelReplaceRequest class to cancel-and-replace an order
  - And, more ...
- Message types are implemented using inheritance.
- Common members (data and function) implemented in Message base class.
- Specific message types are derived from Message base class.
- Some examples shown next:

NewOrderSingle

- A new order is implemented by NewOrderSingle class QuickFIX.

```
class NewOrderSingle : public Message
{
  public:
     .....
     .....
};
```

OrderCancelRequest

- A new order is implemented by OrderCancelRequest class QuickFIX.

```
class OrderCancelRequest : public Message
{
  public:
     .....
     .....
};
```

OrderCancelReplaceRequest

- A new order is implemented by OrderCancelReplaceRequest class QuickFIX.

```
class OrderCancelReplaceRequest : public Message
{
  public:
     .....
     .....
};
```

FIX Application Class

- Any QuickFIX application should implement the `Application` abstract class:

```
class Application
{
public:
 ....
  // Notification of app message being received from target
  virtual void fromApp (const Message&,
        const SessionID& ) = 0;

  ....
};
```

FIX Versions

- First introduced in 1992.
- The FIX protocol has changed over the years to address the ever growing needs of electronic trading.
- New changes have been introduced using different FIX protocol versions.
- QuickFIX uses namespaces to handle versions.

NewOrderSingle

- A new order is implemented by NewOrderSingle class QuickFIX.

```
namespace FIX40
{
    class NewOrderSingle : public Message
    {
        .....
    };
}
namespace FIX42
{
    class NewOrderSingle : public Message
    {
        .....
    };
}
namespace FIX44
{
    class NewOrderSingle : public Message
    {
        .....
    };
}
```

NewOrderSingle

- Suppose we want to use FIX 4.2:

```
FIX42::NewOrderSingle newOrder;

or,

using FIX42::NewOrderSingle;
NewOrderSingle newOrder;
```

- Suppose we want to use FIX 4.4:

```
FIX44::NewOrderSingle newOrder;
```

Next Steps

Next Steps

- We used carefully selected set of problems to practice what we discussed:
  - Some of them are assignments (graded).
  - Some of them are exercises/homeworks (not graded).
  - Coursework include all of them. Practice all of them.
  - Some topics/problems are very broad, e.g. sorting, option-pricers.
  - You should have sufficient coding exercises for at least several more months (e.g. implement new sorting and any other alrogithms; new pricing models such as stochastic vols, jumps etc.).
  - You have sufficient (C++) tools to implement them.
  - If you need any help, do not hesitate to contact me.

My Next Course: Summer

- Some of you asked me about my next course.
- I teach another course in summer quarter:
  - Build on material introduced in this course and introduce high performance computing.
  - Using C++ mostly and some Python.
  - Contact me for more info.

Some Previous Student Feedback (about the HPC course)

- *Recently I joined as a quantitative risk analyst at — this January after graduation. And I feel that the concepts and training in HPC class including multithreading and exposure to linux environment is helpful to my full time job. And I have left reviews on quantnet on January 15 because the HPC course is really helpful.*

- *Currently I am working as a pricing quant for exotics, and applying HPC concepts to my work and really made my life easier.*
  And by the way, this is probably the top three most useful courses I have ever taken from the program (from the POV of a quant who runs lots of Monte Carlo)!

- *Course contents about parallel computing and multithreading are highly related to my summer internship.*

- *After discussing the concepts taught in this class with other FinMath students not enrolled in this class, they wished they would have taken this class and admitted that the material taught would have directly helped them in their summer jobs/internships.*

Course Wrap-up

From "Course Introduction"

- We learn:
  1. C++ and programming concepts
  2. good practices
  3. how to use C++ to write applications in finance
- *Problem driven* approach to introduce various features of C++ in this course.

C++

- C++
  - Tools:
    - IDE
    - the debugger
    - build tools: compiler, preprocessor, linker
    - unit testing framework
  - Functions
    - how to write and use them
    - why we use them
  - Control structures
  - OOP:
    - classes and objects
    - encapsulation and data abstraction
    - inheritance
    - polymorphism
  - Templates
  - C++ Standard Library

Applications

- Example Applications:
  1. Currency converter
  2. Black Scholes analytical pricer
  3. Monte Carlo pricer
  4. Binomial Tree pricer
- Used them to illustrate:
  - Writing correct, efficient, readable, reusable, maintainable and extensible code
  - Refactoring
  - Coding style
  - Unit testing

PLEASE KEEP IN TOUCH

GOOD LUCK!

THANK YOU!