Object Oriented Programming in C++ with Real Example

# MQF 633 C++ for financial engineering

# Objective

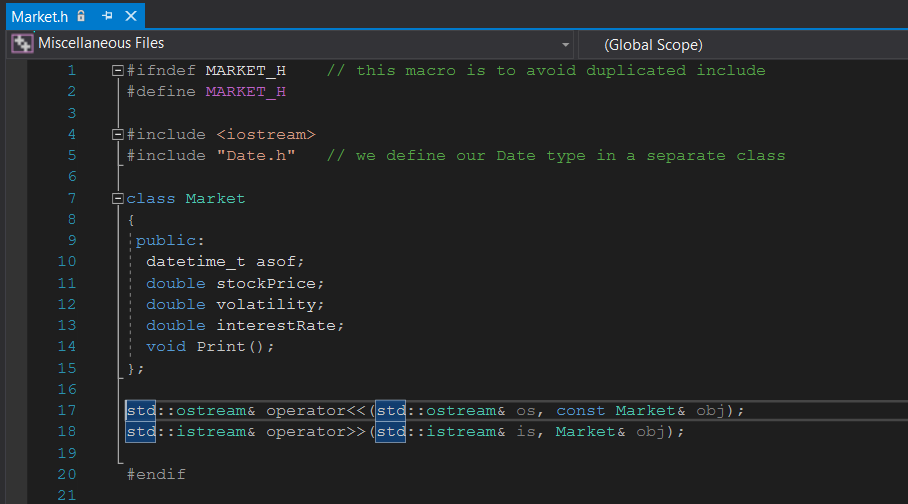
## Get familiar with OOP with some real-world project

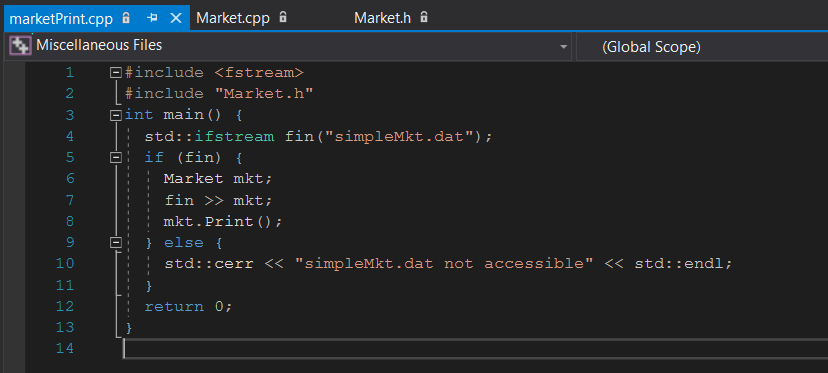
## Get some understanding of some advanced concept

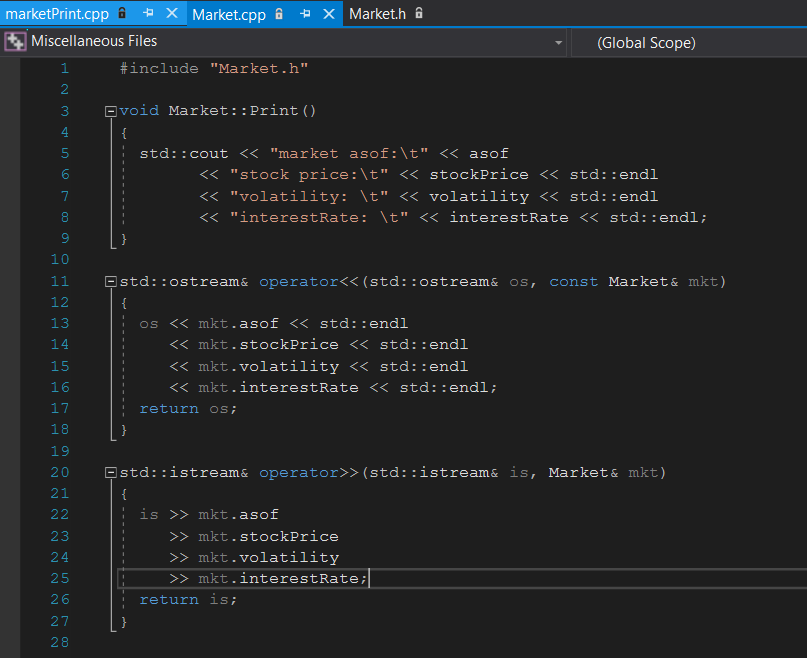
## Brief the first project build a Binomial Tree Pricer to price Option

# Part I: Simple version of Binomial Tree Pricer

# Market Class: refer to file Market.h, Market.cpp and MarketPrint.cpp







* The #ifndef #define #endif macros avoid duplicated declaration when included by other files. This is similar to #program once, which is widely used but not in C++ standard (yet).
* Member function of a class manipulates it’s member variables, including public, protected and private members
* We would like a member function Print() to display the market information
* We would like to overload the >> and << operators so that our market is stream-able
* Market.h -> declaration, Market.Cpp implementation, good practice to separate declaration and implementation for larger project. Why?
* Notice on Operator Overloading:
  + Looks strange at first sight. It is the same as function overloading: reuse the same function name but takes a different set of argument types. Can be used in the normal function call form

(operator>>) (fin, mkt);

or the infix form:

fin >> market; // firstArg operator secondArg

The keyword operator simply make the latter possible

# Default Constructor

A default constructor is a constructor with no arguments

Declaration: ClassName ();

Definition:

ClassName::ClassName () {...}; or

ClassName::ClassName () : m\_A(..), m\_B(..) {...};

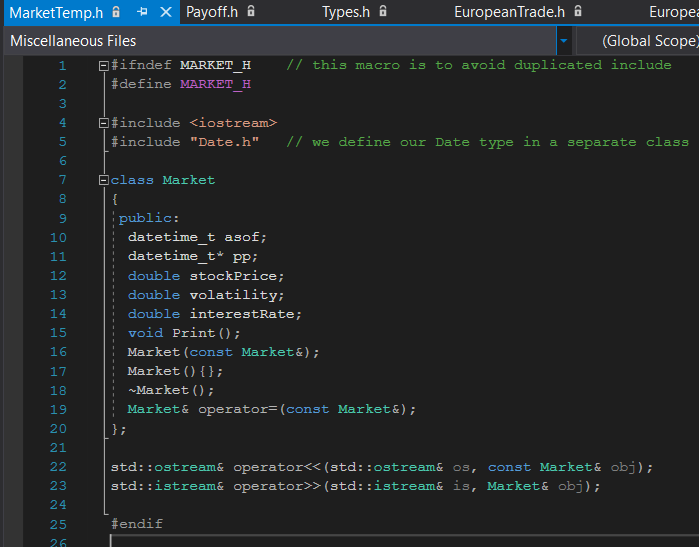
Either defined with an empty parameter list, or with default arguments provided for every parameter

If there is no user-defined constructor, the compiler will always declare a dummy default constructor as an inline public member of its class, e.g., our Market class

When is default constructor called? Market mkt;

# Copy Constructor

Let have a look at another example of market class declaration. Refer to MarketTemp.h



Copy constructor is a constructor and Its parameter is T& (seldom used, why?), or const T&, where T is the type of the class

Declaration: ClassName (const ClassName&);

Definition: ClassName (const ClassName&) { // some code here }

It is used to

* Initialize one object from another of the same type
* Copy an object to pass it as an argument to a function
* Copy an object to return it from a function

When it is used?

* *Direct initialization*: T a(b);, where a and b are both of type T
* *Assignment initialization*: T a = b;

It is often have similar usage of operator overloading const T& operator=(const T& other) {}

**If there is no user defined copy constructor, compiler will make a default one: copy everything by value — not desirable for pointers. Here is an example.**

#include "MarketTemp.h"

Market::Market(const Market& mkt)

{

asof = mkt.asof;

pp = new datetime\_t(\*mkt.pp);

stockPrice = mkt.stockPrice;

volatility = mkt.volatility;

interestRate = mkt.interestRate;

}

Market& Market::operator=(const Market& mkt)

{

if (this == &mkt)

return \*this;

asof = mkt.asof;

if (pp == NULL)

pp = new datetime\_t(\*mkt.pp);

else

\*pp = \*mkt.pp;

stockPrice = mkt.stockPrice;

volatility = mkt.volatility;

interestRate = mkt.interestRate;

return \*this;

}

Key points:

1. When we have a pointer as a member variable: default copy constructor will copy the address the pointer is pointing to, it end up to have a new pointer point to same underlying resource in the memory.
2. Have 2 pointer point to same resource is a very bad practice, why? Recall the pointer delete mechanism

Refer to copyCtor.cpp for more details

#include <fstream>

#include "Market.h"

int main() {

std::ifstream fin("simpleMkt.dat");

if (fin) {

Market mkt;

fin >> mkt;

mkt.Print();

Market mkt2 = mkt;

mkt2.Print();

}

else {

std::cerr << "simpleMkt.dat not accessible" << std::endl;

}

return 0;

}

Refer to copyCtor2.cpp

#include <fstream>

#include "MarketTemp.h"

int main() {

std::ifstream fin("simpleMkt.dat");

if (fin) {

Market mkt;

fin >> mkt;

std::cout << "Market1--------" << std::endl;

mkt.pp = new datetime\_t;

mkt.pp->year = 2010;

mkt.pp->month = 1;

mkt.pp->day = 1;

mkt.Print();

Market mkt2 = mkt;

std::cout << "Market2--------" << std::endl;

mkt2.Print();

std::cout << mkt2.pp->year << std::endl;

Market mkt3;

std::cout << mkt3.pp << std::endl;

mkt3 = mkt2;

std::cout << mkt3.pp << std::endl;

mkt3 = mkt3;

std::cout << mkt3.pp << std::endl;

std::cout << "Market3--------" << std::endl;

mkt3.Print();

std::cout << mkt3.pp->year << std::endl;

}

else {

std::cerr << "simpleMkt.dat not accessible" << std::endl;

}

return 0;

}

# Destructor — Example refer to MarketTemp.cpp

Market::~Market()

{

if (pp != NULL)

delete pp;

}

Revisit: This is exception safe — destructor of an object will always be called when the object goes out of scope or deleted

The check against NULL is to avoid deleting uninitialized pointer, e.g., through other constructors

# C++ Programming Rule of Three

The following three special member functions always come together:

* Copy constructor
* Destructor
* Copy assignment operator =

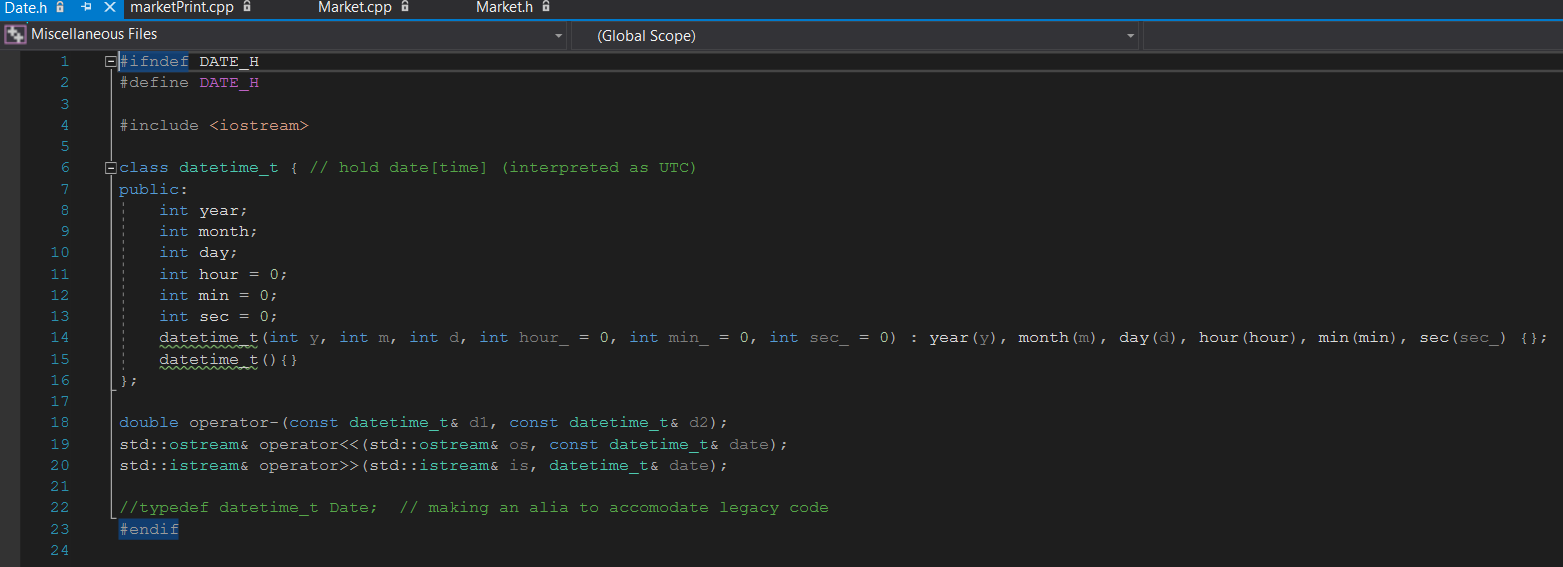
Similar to copy constructor, but called when assigning value of one variable to another existing variable

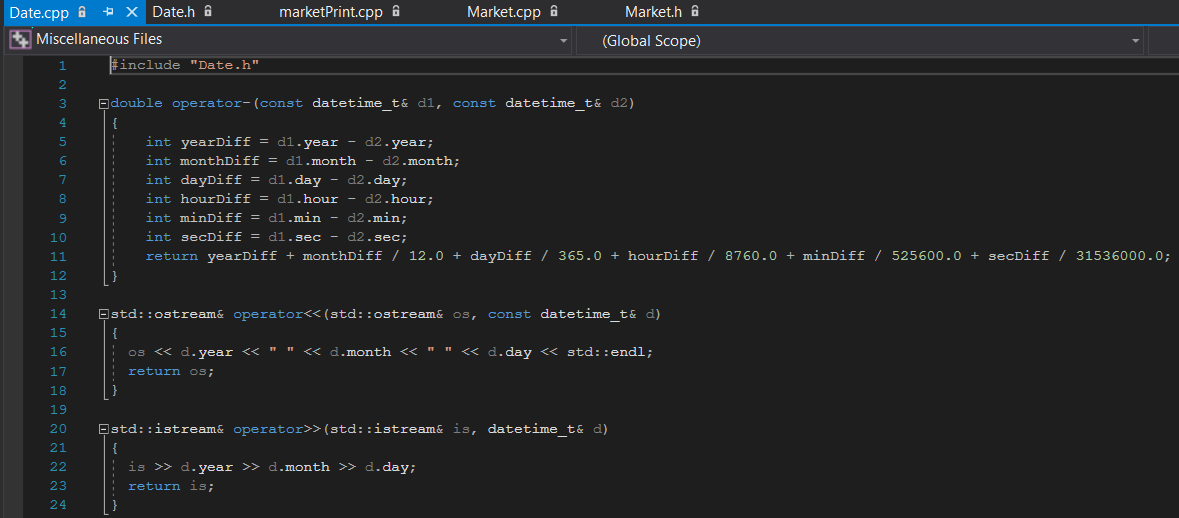
Note the difference of

|  |
| --- |
| T a = b; // copy constructor is called,  // same as T a(b);  T c; c = b; // copy assignment operator is called |

They all have a default version by the compiler, when one needs to be customized, all of the three need to be customized

# Date Class: refer to file Date.h, Date.cp





Another simpler version of this class can be implemented as following.

#ifndef DATE\_H

#define DATE\_H

#include <iostream>

class Date

{

public:

int year;

int month;

int day;

Date(int y, int m, int d) : year(y), month(m), day(d) {};

Date(){};

};

double operator-(const Date& d1, const Date& d2);

std::ostream& operator<<(std::ostream& os, const Date& date);

std::istream& operator>>(std::istream& is, Date& date);

#endif

# Tree class: refer to TreeProduct.h

Now let see an example of using this Date class object:

#ifndef \_TREE\_PRODUCT\_H

#define \_TREE\_PRODUCT\_H

#include "Date.h"

class TreeProduct

{

public:

virtual const datetime\_t& GetExpiry() const = 0;

virtual double Payoff(double stockPrice) const = 0;

virtual double ValueAtNode(double stockPrice, double t, double continuationValue) const = 0;

};

#endif

There is no implementation of class, what does it mean?

The virtual and const=0 make this class an abstract class, which means the function should be implemented in derived class, otherwise there will be compiling error if there is any line of code want to instantiate any real object of TreeProduct class or derived class.

# EuropeanTrade class: refer to file European.h

Now let’s see the example of how derived class is implemented.

#ifndef \_EUROPEAN\_TRADE

#define \_EUROPEAN\_TRADE

#include <cassert>

#include "TreeProduct.h"

#include "Payoff.h"

#include "Types.h"

// still an abstract class?

class EuropeanTrade : public TreeProduct

{

public:

virtual double ValueAtNode(double S, double t, double continuation) const

{ return continuation; }

};

class EuropeanOption : public EuropeanTrade

{

public:

virtual double Payoff(double S) const

{

return PAYOFF::VanillaOption(optType, strike, S);

}

virtual const datetime\_t& GetExpiry() const

{ return expiryDate; }

EuropeanOption(OptionType \_optType, double \_strike, const datetime\_t& \_expiry)

: optType(\_optType), strike(\_strike), expiryDate(\_expiry) {}

private:

OptionType optType;

double strike;

datetime\_t expiryDate;

};

class EuroCallSpread : public EuropeanTrade

{

public:

virtual double Payoff(double S) const

{

return PAYOFF::CallSpread(strike1, strike2, S);

}

virtual const datetime\_t& GetExpiry() const

{ return expiryDate; }

EuroCallSpread(double \_k1, double \_k2, const datetime\_t& \_expiry)

: strike1(\_k1), strike2(\_k2), expiryDate(\_expiry)

{

assert(\_k1 < \_k2);

};

private:

double strike1;

double strike2;

datetime\_t expiryDate;

};

#endif

# Writing Trades As Classes

We now declare our European option:

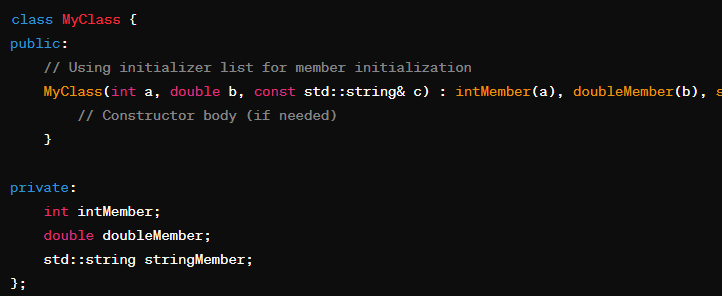


oop/EuropeanTrade.h

A few key points here:

1. Initializer list, why?
   1. **Performance**: Using initializer lists can lead to better performance in certain scenarios. When you use an initializer list, the compiler can often optimize the construction of objects or elements more effectively than if you were to use assignment in the constructor body.
   2. **Avoiding Default Construction**: In some cases, using initializer lists allows you to avoid default construction of objects, which can be useful when default construction is expensive or unnecessary.
   3. **Immutable Members**: If you have **const** or reference members in your class, initializer lists are the only way to initialize them, as they cannot be assigned to after construction.
   4. **Member Initialization Order**: The order in which members are initialized matters in C++. Initializer lists allow you to specify the order explicitly, which can help avoid potential issues related to the order of member initialization.
   5. **Aggregate Initialization**: Initializer lists can be used for aggregate initialization of arrays and structures, making the code more concise and readable.

Example:



1. Is EuropeanTrade a complete class or abstract class, i.e., can we create an object of this class such as:

EuropeanTrade\* eurTrade = new EuropeanTrade();

1. Pay attention to the virtual key word, and function overridden along the hierarchy of classes.
2. Pay attention to the public key word in class inheritance

Class Member Accessibility

|  |  |  |  |
| --- | --- | --- | --- |
| Members | Self | Derived Class | Other Class |
| public | ✓ | ✓ | ✓ |
| protected | ✓ | ✓ | × |
| private | ✓ | × | × |

1. All our member variables are private: we do not want a trade to be changed after it is defined / booked
2. Why we are making Payoff() function a const:
   1. a const function is not allowed to change any of its members
   2. Discussion point: why we want const?
3. Pay attention to the function return types of const type&

# Some other files needed: refer to type.h and payoff.h

#ifndef TYPES\_H

#define TYPES\_H

enum OptionType {Call, Put, BinaryCall, BinaryPut};

#endif

#ifndef PAYOFF\_H

#define PAYOFF\_H

#include "Types.h"

namespace PAYOFF

{

double VanillaOption(OptionType optType, double strike, double S)

{

switch (optType)

{

case Call:

return S > strike ? S - strike : 0;

case Put:

return S < strike ? strike - S : 0;

case BinaryCall:

return S >= strike ? 1 : 0;

case BinaryPut:

return S <= strike ? 1 : 0;

default:

throw "unsupported optionType";

}

}

double CallSpread(double strike1, double strike2, double S)

{

if (S < strike1)

return 0;

else if (S > strike2)

return 1;

else

return (S - strike1) / (strike2 - strike1);

}

}

#endif

Class BinomialTree

#ifndef \_TREE\_PRICER

#define \_TREE\_PRICER

#include <vector>

#include <cmath>

#include "TreeProductNonVirtual.h"

double binomialTree (const Market& market, const TreeProduct& trade, int N)

{

double T = trade.GetExpiry() - market.asof;

double sigma = market.volatility;

double rate = market.interestRate;

std::vector<double> states(N+1);

double dt = T / N;

double b = std::exp((2\*rate+sigma\*sigma)\*dt)+1;

double u = (b + std::sqrt(b\*b - 4\*std::exp(2\*rate\*dt))) / 2 / std::exp(rate\*dt);

double p = (std::exp(rate\*dt) - 1/u) / (u - 1/u);

// initialize the final states, apply payoff directly

for (int i = 0; i <= N; i++) {

double S = market.stockPrice \* std::pow(u, N-2\*i);

states[i] = trade.Payoff(S);

}

for (int k = N-1; k >= 0; k--)

for (int i = 0; i <= k; i++)

states[i] = states[i]\*p + states[i+1]\*(1-p);

return exp(-rate\*T) \* states[0];

}

#endif

Discussion point:

1. Why parameter list using const Market&, and const TreeProduct& ?
2. Pay attention to STL template class std::vector<>
3. A little bit on binomial tree setup

Now, let’s put a few things we have gone through into a simple program.

#include <fstream>

#include "Market.h"

#include "TreePricer.h"

#include "EuropeanTrade.h"

int main()

{

std::ifstream fin("simpleMkt.dat");

if (fin) {

Market mkt;

// fin >> mkt;

fin >> mkt;

mkt.Print();

EuropeanOption callOption1(Call, 90, datetime\_t(2016, 1, 1));

EuropeanOption callOption2(Call, 91, datetime\_t(2016, 1, 1));

EuroCallSpread callSpread(90, 91, datetime\_t(2016, 1, 1));

std::cout << binomialTree(mkt, callOption1, 100) << std::endl;

std::cout << binomialTree(mkt, callOption2, 100) << std::endl;

std::cout << binomialTree(mkt, callSpread, 100) << std::endl;

std::cout << binomialTree(mkt, callOption1, 100) - binomialTree(mkt, callOption2, 100) << std::endl;

return 0;

}

else {

std::cerr << "no simpleMkt.dat found" << std::endl;

return 1;

}

}

Simple right? We created 1 market data object, flush its value using DAT file, and then created 3 option trade objects. Finally we call the simple binominal Tree function to compute the option value.

So far we have below in class inheritance

TreeProduct

EuroCallSpread

EuropeanOption

BinomialTree()

Trade object

Market object

Any problem from above setup? Extended discussion.

1. What about there are other option type?
2. What about the current simple version of binomial tree is not good enough, how we extend out code easily to support it.

# Part II: Extension

Add American trade and two types of American option into curent Tree product hierachy.

TreeProduct

AmericanTrade

EuropeanTrade

AmerCallSpread

AmericanOption

EuroCallSpread

EuropeanOption

class AmericanTrade : public TreeProduct

{

public:

virtual double ValueAtNode(double S, double t, double continuation) const

{ return std::max(Payoff(S), continuation); }

};

class AmericanOption : public AmericanTrade

{

public:

virtual double Payoff(double S) const

{

return PAYOFF::VanillaOption(optType, strike, S);

}

virtual const Date& GetExpiry() const

{

return expiryDate;

}

AmericanOption(OptionType \_optType, double \_strike, const Date& \_expiry)

: optType(\_optType), strike(\_strike), expiryDate(\_expiry) {}

private:

OptionType optType;

double strike;

Date expiryDate;

};

class AmerCallSpread : public AmericanTrade

{

public:

virtual double Payoff(double S) const

{

return PAYOFF::CallSpread(strike1, strike2, S);

}

virtual const Date& GetExpiry() const

{

return expiryDate;

}

AmerCallSpread(double \_k1, double \_k2, const Date& \_expiry)

: strike1(\_k1), strike2(\_k2), expiryDate(\_expiry)

{

assert(\_k1 < \_k2);

};

private:

double strike1;

double strike2;

Date expiryDate;

};

Discussion point:

* What is American option?
* Can we price the American option in the same way of European option? If not, what would be the procedure tree pricer need to have?

# Pricing American Option With Tree

Tree was not invented for European payoff. There are plenty of more efficient methods for pricing European products

* Closed-form solution (analytic pricer)
* Numerical integration (semi-analytic)

Tree is the most natural way to handle American payoff due to the backward iteration. At any point in time, or any node of the tree, we know the continuation value of the product — through calculating the conditional expectation

American product allows the option holder to exercise the option any time

This translates to the choice to make at any node of the tree: continue holding the option or take the intrinsic value

* Continue holding the option — the option worth its conditional expectation
* Exercise now — the option worth its intrinsic value

Optimal exercise strategy is to exercise when intrinsic value worth more than the continuation value — taking the max

# Steps on price American option on binomial tree

Here are the steps involved in pricing an American option using a binomial tree:

1. Set up the Binomial Tree:

* Define the parameters: underlying asset price (S), strike price (K), time to expiration (T), volatility (σ), risk-free interest rate (r), and the number of time steps (n).
* Determine the time step size (Δt) by dividing the total time to expiration (T) by the number of time steps (n).

1. Calculate Up and Down Movements and probability

* Calculate the up factor (u) and down factor (d) based on the volatility and time step size.



*  Calculate probability of each node.

1. At last time step T, calculate option value as excised payoff. Then at t = T- Δt, it value is
2. Early Exercise:

At each node of t = T – i\*Δt , compare the calculated option value with the intrinsic value (the difference between the current stock price and the strike price). If the intrinsic value is higher, the option holder may choose to exercise early.

V(t - Δt ) = Max(V(t) \* Df, exercised value)

1. Backward Induction: Move backward through the tree, updating option values based on the early exercise decisions and the option pricing formula.
2. Final Result: The option value at the starting node of the tree is the estimated fair price for the American option.

First step, we can simply rewrite the bimoinal function to handle the American option.

double binomialTreeAmer (const Market& market, const TreeProduct& trade, int N)

{

// Set up the tree

...

// initialize the final states, apply payoff directly

for (int i = 0; i <= N; i++) {

double S = market.stockPrice \* std::pow(u, N-2\*i);

states[i] = trade.Payoff(S);

}

for (int k = N-1; k >= 0; k--) {

for (int i = 0; i <= k; i++) {

double S = market.stockPrice \* std::pow(u, k-2\*i);

double df = exp(-rate\*dt);

double continuation = df \* (states[i]\*p + states[i+1]\*(1-p));

states[i] = std::max( continuation, trade.Payoff(S) );

}

}

return states[0];

}

Secondly, let extend our BinomialTree function into a class for a better code.

#include <vector>

#include <cmath>

#include "TreeProduct.h"

class BinomialTreePricer

{

public:

BinomialTreePricer(int N) {

nTimeSteps = N;

states.resize(N+1);

}

double Price(const Market& mkt, const TreeProduct& trade)

{

// model setup

double T = trade.GetExpiry() - mkt.asof;

double dt = T / nTimeSteps;

ModelSetup(mkt.stockPrice, mkt.volatility, mkt.interestRate, dt);

// initialize

for (int i = 0; i <= nTimeSteps; i++) {

states[i] = trade.Payoff( GetSpot(nTimeSteps, i) );

}

// price by backward induction

for (int k = nTimeSteps-1; k >= 0; k--)

for (int i = 0; i <= k; i++) {

// calculate continuation value

double df = exp(-mkt.interestRate\*dt);

double continuation = df \* (states[i]\*GetProbUp() + states[i+1]\*GetProbDown());

// calculate the option value at node(k, i)

states[i] = trade.ValueAtNode( GetSpot(k, i), dt\*k, continuation);

}

return states[0];

}

protected:

virtual void ModelSetup(double S0, double sigma, double rate, double dt) = 0;

virtual double GetSpot(int ti, int si) const = 0;

virtual double GetProbUp() const = 0;

virtual double GetProbDown() const = 0;

private:

int nTimeSteps;

std::vector<double> states;

};

And we further extend this class into a class with CIR model.

class CRRBinomialTreePricer : public BinomialTreePricer

{

public:

CRRBinomialTreePricer(int N) : BinomialTreePricer(N){}

protected:

virtual void ModelSetup(double S0, double sigma, double rate, double dt)

{

double b = std::exp((2\*rate+sigma\*sigma)\*dt)+1;

u = (b + std::sqrt(b\*b - 4\*std::exp(2\*rate\*dt))) / 2 / std::exp(rate\*dt);

p = (std::exp(rate\*dt) - 1/u) / (u - 1/u);

currentSpot = S0;

}

virtual double GetSpot(int ti, int si) const

{

return currentSpot \* std::pow(u, ti-2\*si);

}

virtual double GetProbUp() const {return p;}

virtual double GetProbDown() const {return 1-p;}

private:

double u; // up multiplicative

double p; // probability for up state

double currentSpot; // current market spot price

};

To note:

* Below function is declared in class of BinomialTreePricer, but not implemented. This makes the base class of BinomialTreePricer as an interface class. It means we cannot create an object on this class alone.

virtual void ModelSetup(double S0, double sigma, double rate, double dt) = 0;

virtual double GetSpot(int ti, int si) const = 0;

virtual double GetProbUp() const = 0;

virtual double GetProbDown() const = 0;

* These function implementation in derived class, and gives the details of how the Tree should be built within this model. With this design, we can easily extend base tree model into different types of model if the only difference here is how to build a Tree.
* Pay attention to ValueAtNode():
  + This is declared as pure virtual function at class TreeProduct level.
  + The pure virtual function ValueAtNode() forces us to implement it in all of its derived classes
  + This is good since it provide an common interface but maybe different behaviour for different product

class EuropeanTrade : public TreeProduct

{

virtual double ValueAtNode( double S, double t, double continuation) const { return continuation; }

};

class EuropeanOption : public EuropeanTrade

{ ... };

class EuropCallSpread : public EuropeanTrade

{ ... };

class AmericanTrade : public TreeProduct

{

virtual double ValueAtNode( double S, double t, double continuation) const { return std::max(Payoff(S), continuation); }

};

class AmericanOption : public AmericanTrade

{ ... };

class AmerCallSpread : public AmericanTrade

{ ... };

# Look at our tree algorithm again

Algorithm 1 Tree Pricer

1: Set up the tree and parameters

2: Initialize the last time slice with final payoff

3: for *k* = *N* − 1 to 0 do

4: for *i* = 0 to *k* do

5: Calculate the continuation value (discounted expectation)

6: Given the information of the tree node, calculate the option value at Node(k, i)

7: end for

8: end for

Step 5 belongs to the tree pricer

Step 6 is fully determined by the product:

Input: stock price *S*, continuation value *V*, current time *t*

Output: current option value

Now we can use below pseudo-code to test the new tree pricer on both product.

void testAmer()

{

// add code of load market//

size\_t treeTimeSteps = 200;

for (int i = 0; i <= treeTimeSteps; i++) {

EuropeanOption euroCall(Call, i, Date(2016, 1, 1));

AmericanOption amerCall(Call, i, Date(2016, 1, 1));

BinomialTreePricer\* ePricer = new CRRBinomialTreePricer(treeTimeSteps);

BinomialTreePricer\* aPricer = new CRRBinomialTreePricer(treeTimeSteps);

ePricer->ModelSetu( … );

aPricer->ModelSetu( … );

}

std::cout << ePricer->price( …, … ) << "\t\t\t"<< aPricer->price(…, …) << std::endl;

}

# Trade example 1Y call option *N* = 100, *S*0 = 100, *r* = 3%, *σ* = 10%

### American Call versus European Call

−

20

0

20

40

60

80

100

120

140

160

180

200

220

0

50

100

European call price

American call price

Strike

* American call and European call prices are identical
* It is never optimal to early exercise a call option (under which condition?)

### American Put versus European Put

−

20

0

20

40

60

80

100

120

140

160

180

200

220

0

50

100

European put price

American put price

Strike

American put price is above European put price and it is possible to early exercise

# Include Chart — main.cpp

main.cpp

fstream

Market.h

TreePricer.h

EuropeanTrade.h

AmericanTrade.h

iostream

Date.h

vector

cmath

TreeProduct.h

cassert

Payoff.h

Types.h

# CRR Binomial Tree Models

Recall that we use CRR binomial tree model [1] where the additional constraint *u* = *d*1 was imposed to restrict the solution space when trying to match the second moment

Recall also that the first moment is matched by setting

# JRRN Binomial Model

The constraint is arbitrary, as long as first and second moment matches

Jarrow et al. [2] proposed making

* It can be verified that the second moment matches
* Note that the original Jarrow Rudd (JR) tree enforces the probability to 0.5 and is thus not a risk neutral tree (used for credit risk where the market is not complete)

# BinomialTreePricer Class

* There can be more ways to impose the constraint, see [3] for 11 methods on setting up the binomial tree parameters
* The general purpose is to tune the geometry of the binomial tree to favour the problem at hand
* Different binomial trees only differ at the tree geometry — candidate for abstraction

# Binomial Tree Pricer Hierarchy

BinomialTreePricer

+

BinomialTreePricer

()

+

()

Price

# ModelSetup()

# GetSpot()

# GetProbUp()

# GetProbDown()

CRRBinomialTreePricer

+

CRRBinomialTreePricer

()

# ModelSetup()

# GetSpot()

# GetProbUp()

# GetProbDown()

JRRNBinomialTreePricer

+

JRRNBinomialTreePricer

()

# ModelSetup()

# GetSpot()

# GetProbUp()

# GetProbDown()

# Project of first half

## Create a project of windows or OS console application, where contains the class of market, date, American option trade, European option trade

## Implement a simple function version of Binomial trade pricer for European option, and price a European stock call option with T = 3 month, R = 4%, vol = 15%. Choose simple version of risk free pricing on Binomial tree.

## Implement a simple function version of Binomial trade pricer for American option, and price an American stock call option with T = 3 month, R = 4%, vol = 15%. Choose simple version of risk free pricing on Binomial tree.

## Extend binomial tree function into class implementation, and price the same European and American option

# References

John C. Cox, Stephen A. Ross, and Mark Rubinstein.



Option pricing: A simplified approach.

*Journal of Financial Economics*, 7(3):229–263, 1979.

Robert Jarrow and Stuart M Turnbull.



Pricing derivatives on financial securities subject to credit risk. *Journal of Finance*, 50(1):53–85, 1995.

Mark S. Joshi.



The convergence of binomial trees for pricing the american put. 2007.