FINM 326: Computing for Finance in C++

Lecture 8

Chanaka Liyanarachchi

February 24, 2023

Black Scholes Pricer: Revisit

Polymorphism

Monte Carlo Pricer

Distributions

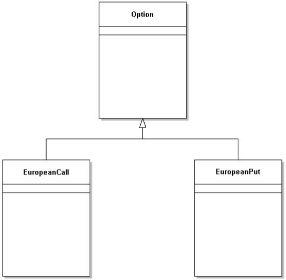
More on Inheritance

C++ Standards

Black Scholes Pricer: Revisit

Black Scholes Pricer: Option Classes

- Last week we introduced the following class design to represent Options:



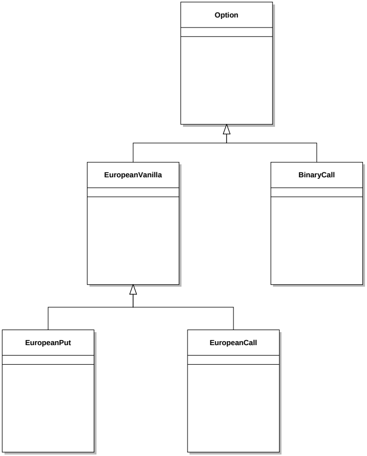
- This design has some advantages:
  - reusing code
  - extensible

Black Scholes Pricer: Pricing Methods

- The Price and Delta functions in the (Option) base class are pure virtual:
  - No common default implementation that works for every option type.
  - Each derived class has a specialized implementation.
- What about Gamma? it is the same for (European Vanilla) Calls and Puts.
- Choices:
  1. non virtual: define and implement in the base class
  2. virtual: base class provides a default implementation
  3. pure virtual: each derived class implements it
- Note: Any choice is acceptable for Assignment 4.

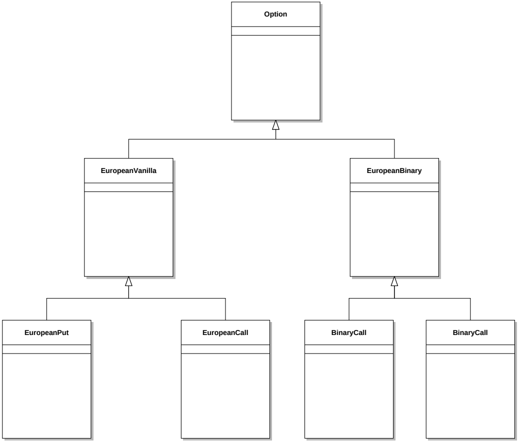
- Choices 1 and 2: base class has to provide an implementation.
- One argument: that's ok because Calls and Puts have same gamma.
- Another argument: that's not true all options (e.g. binary options).
- Two option types have a common greek doesn't mean all option types will have the same greek.
- If we use Choice 3: we are repeating the same code in Call and Put option classes.

- We can use a slightly different class hierarchy:



- Gamma pure virtual in the base (Option) class.
- Implement the common gamma for vanilla European options in `EuropeanVanilla` class.
- We reuse code but we don't have to provide a default implementation for all types.

- We can make our design even more flexible:



- Exercise: Extend Black Scholes pricer to support binary (digital) Options `https://en.wikipedia.org/wiki/Binary_option`.

Polymorphism

Polymorphism

- The types related by inheritance are known as polymorphic[1]. types.
- We can use the polymorphic types interchangeably.
- Polymorphic behavior applies to a reference or a pointer to a polymorphic type only.
- We can use **a pointer or a reference** to a base class object to point/bind to an object of a derived class – this is known as the Liskov Substitution Principle (LSP).

[1] derived from a Greek word which means *many forms*

Using Polymorphic Types: Example 1

- Option types are polymorphic.
- We can use a pointer to a `Option` to point to any of the derived option types:

```
Option* option1 =
      new EuropeanCall(/*parameters omitted*/);

or,

Option* option1 =
     new EuropeanPut(/*parameters omitted*/);
```

Using Polymorphic Types: Example 2

- We can also use a reference to a `Option` to bind to any of the derived Option types:

```
EuropeanCall call(/*parameters omitted*/);
Option& option1 = call;

or,

EuropeanPut put(/*parameters omitted*/);
Option& option1 = put;
```

- Suppose we invoke a virtual (or pure virtual) member function (e.g. `Price()`), using such a reference/pointer:

```
option1->Price(/*parameters omitted*/);  //Example 1

or,

option1.Price(/*parameters omitted*/); //Example 2
```

- The member function based on the type of the object the pointer points (or the reference is bound) to will be used.
- For example, if option1 is pointer/reference to a:
  - `EuropeanCall`: then `EuropeanCall::Price()` will be used
  - `EuropeanPut`: then `EuropeanPut::Price()` will be used
- This allows us to write flexible code using base class types and without needing to know the derived type of an object.

Using Polymorphic Types: Example 3

- Suppose we have an option portfolio of Calls and Puts. How do we store them in a container:
  ```
  vector<EuropeanCall> calls;
  vector<EuropeanPut> puts;
  ```
- We can write a function to find the value of the portfolio:
  ```
  double PortfolioValue(const vector<EuropeanCall>& calls)
  {
      //for each option in calls price it
      //add price to sum
      //return sum
  }
  ```
- We have to write a similar function for EuropeanPuts. Not extensible.

- Options are polymorphic:

```
vector<Option*> calls_and_puts;
```

- Now, we can write one function to handle all option types:

```
double PortfolioValue(const vector<Option*>& options)
{
   //same as before
}
```

OOP: Roadmap

- OOP Main Concepts:
  1. Classes and Objects ✓
  2. Data Abstraction and Encapsulation ✓
  3. Inheritance ✓
  4. Polymorphism ✓

Detour: Const and Mutable Members

Const Member Functions

- We saw how to declare a `const` value in L2:

```
const int PI = 3.14;
```

- We can read it, but we cannot change it.
- Can we create constant objects?

- Suppose we have a class:

```
class Currency
{
public:
    .....
    string GetSymbol();
private:
   string symbol_;
   ...
};

string Currency::GetSymbol()
{
   return symbol_;
}
```

- We can create a `const Student` object:
  `const Currency c("CAD", 1.2);`
- Changing this object (s) is not allowed.
- We should be able to use an operation, e.g. `GetSymbol()` that does not change the object:

```
cout << c.GetSymbol() << endl;
```

- But the above line doesn't compile. What is the problem here?
- GetSymbol() doesn't change the object, but the compiler doesn't know that.

- Programmer needs to use const keyword to specify if a member function doesn't modify class data members.
- A const object can invoke const member functions of that class only.

```
class Currency
{
public:
    //....
    string GetSymbol() const;

};

string Currency::GetSymbol() const
{
   return symbol_;
}
```

- Now we can use this operation on a const Currency object:

```
cout << c.GetSymbol() << endl;
```

- We can overload a function on the basis of const (i.e. write a const and non const versions of the same function).

- When we use inheritance, const keyword has to be applied correctly in all derived types:

```
class Option
{
public:
    //....
    virtual double Price(/*parameters*/) const = 0;

};

class EuropeanCall
{
public:
    //....
    double Price(/*parameters*/) const override;

};

double EuropeanCall::Price(/*parameters*/) const
{
   .......
   .......
}
```

Mutable

- A const member function is not allowed to modify any data member of that class.
- There is an exception to that rule: a data member is marked `mutable` can be modified inside a const member function.

Monte Carlo Pricer

Monte Carlo Pricer

- Today we will use Monte Carlo to price European Options on stocks.
- Objectives:
  1. Illustrate OOP concepts and class design.
  2. Introduce random number generation.
  3. Introduce more math functions.
- References:
  - John Hull. Options Futures and Other Derivatives
  - Paul Glasserman. Monte Carlo Methods in Financial Engineering.

Monte Carlo Technique: Background

- The (time 0) value of an option is the discounted *expectation*[2] of its payoff under the risk neutral measure[3].
- That means: to price an option we need to compute an *expected value*, i.e. an integral.
- Monte Carlo is a *numerical technique* used to **estimate** an integral.

[2] `https://en.wikipedia.org/wiki/Expected_value`

[3] `http://en.wikipedia.org/wiki/Risk-neutral_measure`

Notation

We use the following notation:

S_t	:	Stock price at time t
σ	:	Volatility of the Stock (assumed constant)
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*[4].
- 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:

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

[4] http://en.wikipedia.org/wiki/Geometric_Brownian_motion

- Using equation (4), we can simulate the stock price at any (future) time t.
- $N(0,1)$ is a random observation (random number) drawn from the standard normal distribution.
- We need S_T (i.e. price at time T) to find the payoff of a European Option at Option expiration.
- Let's say, $z =$ a random number drawn from N(0, 1).

$$S_T = S_0 \exp^{((r-\sigma^2/2)T+\sigma z\sqrt{T}} \tag{5}$$

- Now we can find the Option payoff using the appropriate payoff function, e.g. for a Call:

$$(S_T - K)^+ \tag{6}$$

Monte Carlo Option Pricer: Step by Step

- Draw a random number (z_i) from the standard normal distribution, N(0, 1). Here, the subscript i denotes the i^{th} draw.
- Simulate the stock price at the Option expiration (S_T). The stock price at the expiration is given by:

$$S_{T,i} = S_0 \exp^{((r-\sigma^2/2)T+\sigma z_i\sqrt{T}} \tag{7}$$

- For a given price path (i) find the discounted payoff. E.g. European Call payoff at time zero is given by:

$$C_i = \exp^{-rT}(S_{T,i} - K)^+ \tag{8}$$

- Repeat above steps for (M) trials, and calculate the Option payoff for each path.
- Find the mean of the discounted Option payoffs:

$$\hat{C} = \frac{\exp^{-rT}}{M} \sum_{i=1}^{M} (S_{T,i} - K)^{+} \tag{9}$$

- This is an **estimate** of the Option price.
- *The law of large numbers* states that such an estimate converges to the correct value as the number of trials (M) increases.

Monte Carlo Option Pricer: Number of Simulations

- It can be shown that the standard error (ϵ) of this estimate:
  $\epsilon = \frac{\omega}{\sqrt{M}}$
  where, ω is the sample standard deviation.
- This means: $\epsilon \propto \frac{1}{\sqrt{M}}$
- To increase the accuracy of a simulation (i.e. decrease error) X 2, need increase the no of trials X 4 (i.e. 2^2).
- The confidence interval of 100(1-p)% is given by (Central Limit Theorem):
  $$\left(\hat{C} - \Phi^{-1}(1 - \tfrac{p}{2})\tfrac{\omega}{\sqrt{M}}, \hat{C} + \Phi^{-1}(1 - \tfrac{p}{2})\tfrac{\omega}{\sqrt{M}}\right)$$
- E.g. for a 0.95 confidence interval:
  $$\left(\hat{C} - 1.96\tfrac{\omega}{\sqrt{M}}, \hat{C} + 1.96\tfrac{\omega}{\sqrt{M}}\right)$$

Random Numbers

Pseudo Random Numbers

- We need random numbers for Monte Carlo simulations.
- We use algorithms generate *pseudo* random numbers.
- A good algorithm can generate good pseudo random number sequences that satisfy statistical *randomness* properties.

Generating Pseudo Random Numbers

- We have several options to generate random numbers:
  1. Do it ourselves.
  2. Use a random number generator in the Standard Library.
  3. Use a third party library (when performance is critical, summer HPC course).
- Initially we will generate random numbers using a simple algorithm:
  - To understand what's involved
  - Practice programming
  - Use some more math functions in C++

Box-Muller

- There are several simple algorithms:
  - Box-Muller
  - Summation
  - Polar
- We will use the Box-Muller transform algorithm.

- The Box-Muller algorithm works as follows:
  - Draw two independent random numbers, x and y, from the uniform distribution in [0, 1].
  - Find z such that:

$$z = \sqrt{-2\ln{(x)}}cos(2\pi y) \qquad x \neq 0 \tag{10}$$

    z is a standard normal random variable.

Box-Muller: Thinking in C++

- `rand()` a returns an integer (pseudo) random number from the uniform distribution in [0, `RAND_MAX`]:
  - defined in `cstdlib` header `http://www.cplusplus.com/reference/cstdlib/`
  - 0 and `RAND_MAX` included
  - `RAND_MAX` is implementation defined, at least 32767
- To get a uniform (pseudo) random number in [0,1]:

```
double x = static_cast<double>(rand()) /  RAND_MAX;
double y = static_cast<double>(rand()) /  RAND_MAX;
```

- We can use x and y to find z, a pseudo random variable, from the standard normal distribution. Remember, $x \neq 0$.

```
double z = sqrt(-2.0*log(x)) * cos(2*PI*y);
```

Seed

- We use a seed to initialize a pseudo random number generator:
  - same seed to generate the same sequence
  - unique seed to generate a unique sequences
- `srand()` is used to initialize a random number generator with a seed.
- To initialize the seed to using a given integer, e.g. 17:
  `srand(17);`
- To get a unique seed, one common technique is use the current time as the seed:

  ```
  srand(static_cast<unsigned int>(time(0)));
  ```

Simulating the Stock Price Path

- Suppose, the `BoxMuller()` generates a standard normal random number:

```
double z_i = BoxMuller();
```

- Using any i^{th} generated random number z_i, we can simulate a stock price $S_{T,i}$, at time T:

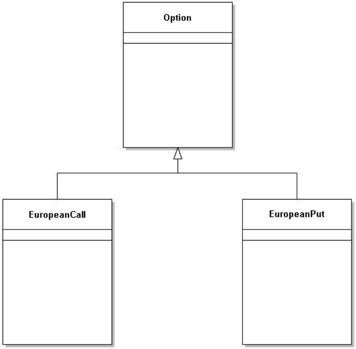
```
double ST_i =
    S0*exp((r-sigma*sigma/2.0)*T + sigma*z_i*sqrt(T));
```

- Now we can estimate an Option price using steps described earlier.

Monte Carlo Pricer

Option Class Hierarchy

We can still use this class design (with some changes):



Extensibility

- Black Scholes pricer design allows us to add new option types easily.
- Option types is not the only thing we want to add.
- An extensible design should allow us to add:
  - new option types (Call, Put etc.)
  - new exercise types (European, American etc.)
  - new models
  - new pricing techniques

- Current design has limited extensibility:
  - Main issue: pricing is tied to the Option class (we have the `Price()` function in the Option class).
  - Doesn't allow us to add new pricing techniques and/or models in an extensible way.
  - One (bad) choice would be to add new price functions for each model/technique in Option class:
    - MCPrice(): to price using Monte Carlo
    - TreePrice(): to price using a tree
  - This is not an extensible design – we won't do that.

MCPricer Class (Incomplete)

- We will move the Price() to a separate class.
- For Monte Carlo, implement pricing in MCPricer class.
- The `Price()`: prices an option using Monte Carlo

```
class MCPricer
{
public:
   double Price(const Option& option,
        double stockPrice,  double vol, double rate,
        unsigned long paths);
};
```

- `Price()`:
  - Takes a reference to an Option.
  - Any derived class type (`EuropeanPut`, `EuropeanCall`) can be used thanks to polymorphism.

- Incomplete Price() implementation:

```
double MCPricer::Price(const Option& option,
      double stockPrice,
      double vol sigma, double rate,
      unsigned long paths)
{
   double T = option.GetTimeToExpiration();

   for (unsigned int i=0; i<paths; ++i)
   {
      generate random number;

      generate stock price (ST)

      payoff = option.GetExpirationPayoff(ST)

      .....
   }
   //calculate price

   return price;
}
```

Option (Base) Class (Incomplete)

- Each option type must support a payoff calculation.
- Depends on the Option type.
- We need to introduce a new pure virtual function to find payoffs:

```
class Option
{
public:
   //.....

   virtual double GetExpirationPayoff(double ST) const = 0;

   //.....
};
```

- It needs to be const member function (it does not change any class data members) as we're using the object as a const reference.

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

```
int main()
{
   MCPricer mc;

   EuropeanCall call(K, T);

   double callPrice = mc.Price(call, S, v, r, M);

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

   EuropeanPut put(K, T);

   double putPrice = mc.Price(put, S, v, r, M);

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

Assignment 5 (Graded Assignment)

- Complete Monte Carlo option pricer using the class design described above.
- Price European vanilla Call and Put Options.
- Use:

  $S_o = 100.0$
  $\sigma = 0.3$
  $r = 0.01$
  $\mathrm{T} = 2.0$
  $\mathrm{K} = 100.0$
  M (Paths) $=$10000, 100000, 1000000
- Submit all code.
- Individual assignment.
- Due: March 3 by midnight CST.

Remarks

- We looked at Monte Carlo technique in its basic form.
- Our main goal today is to show how to apply OOP to solve the basic pricing problem.
- The basic form is computationally **inefficient**.
- There are various techniques to improve the efficiency of a Monte Carlo simulation.
- You can use this program as the starting point to introduce changes/optimizations.

Distributions

Distributions

- We can use the Standard Library to generate random numbers.
- Random number generation is achieved using:
  - generators (engines)
  - distributions
- Generators produce uniform random values using well known algorithms.
- Generators differ in complexity, quality and speed.
- Uniform random values are mapped to random numbers for well known distributions.
- A list of generators and distributions is available at `http://www.cplusplus.com/reference/random/`
- Generators and distributions are defined in `random` header.

- To generate standard normal random numbers:

```
mt19937 e;

e.seed(771);

normal_distribution<double> d(mean, stdev);

for (int i=0; i<5;++i)
{
    cout <<  d(e) << endl;
}
```

- Homework: Redo Assignment 5 using generators in the Standard Library.[5]

[5] You should use Box-Muller for Assignment 5.

More on Inheritance

Virtual Destructor

- Consider the example below:

```
class Base1
{
public:
  virtual void Fun1();

  virtual ~Base1();
};

class Derived1 : public Base1
{
public:
  void Fun1();

  ~Derived1();
};

int main()
{
   Base1* p = new Derived1();

   delete p;
}
```

- When we delete a derived class we should execute the derived class destructor and the base class destructor.
- A virtual base class destructor is needed to make sure the destructors are called properly when a derived class object is deleted through a pointer to a base class.
- If we delete a derived class object through a pointer to a base class when the base class destructor is non-virtual, the result is *undefined*.

Public Inheritance

- The form of inheritance we've seen is known as *public inheriance*.
- It is used to model the *is-a* relationship between classes:
  - Student *is-a* Person; Employee *is-a* Person
  - EuropeanCall *is-an* Option; EuropeanPut *is-an* Option
- Public inheritance is the most widely used form.

Inheritance: Other Forms

- We have 2 other forms of inheritance:
  - private
  - protected
- The meanings of those forms are different from public inheritance.

Multiple Inheritance

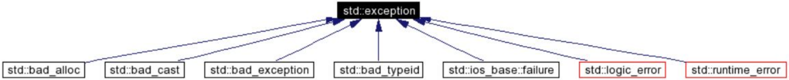
- C++ allows a class to derive from more than one base class.
- It is known as multiple inheritance.
- We will see an examples where multiple inheritance is used when we look at electronic trading.

Inheritance Examples from C++ Standard Library

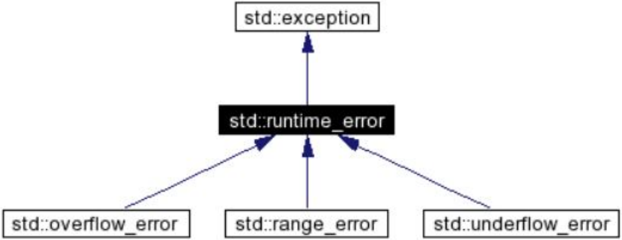
- Inheritance is a widely used concept.
- Shown below are some examples from C++ Standard Library:
    1. Exceptions
    2. Streams

Exception Class Hierarchy

- std::exception:



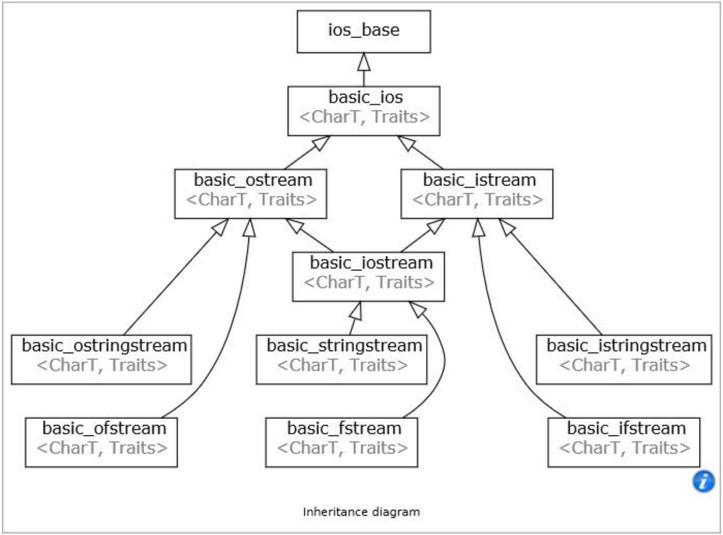
- std::runtime_error:



- `https://en.cppreference.com/w/cpp/error/exception`

Streams Class Hierarchy

- Stream class hierarchy:



Inheritance diagram

- Ref: `https://en.cppreference.com/w/cpp/io`

C++ Standards

- Designed and implemented by Bjarne Stroustrup (Bell Labs) in early 80s.
- C++ is standardized by an ISO (International Organization for Standardization) working group in 1998.
- First standardized C++ was released in 1998 (this release is informally known as C++98).
- Major changes/features introduced in 2011. Known as C++11.
- Since then, new standards introduced in 2014, 2017 and 2020, commonly referred to as C++14, C++17, C++20
- Most commonly used compilers supports most (not all) features from the current standard, and some features from upcoming standards.
- We used various features from recent standards, including some features from the most recent C++20 standard in this course.

C++11

- We've seen some features from C++11:
  - auto
  - uniform_initialization
  - shared_ptr
  - distributions
- And, the following features from C++20:
  - optional
  - numbers (π etc.)