FE545 Design, Patterns and Derivatives Pricing

Bridging with a Virtual Constructor

Steve Yang

Stevens Institute of Technology

steve.yang@stevens.edu

02/06/2024

Overview

The Problem - Vanilla Option

- Options belong to the larger group of securities known as derivatives. A derivative's price is dependent on or derived from the price of something else. Options are derivatives of financial securities, and their value depends on the price of some other asset. Examples of derivatives include calls, puts, futures, forwards, swaps, and mortgage-backed securities, among others.
- A simple solution would be to define a Vanilla Option class which has data members such as PayOff and Expiry, etc. Do we have problem with our existing design?
  - The class PayOff is abstract, and the complier will complain if we want to have a data member for our class.
  - What we are doing is to contain an object from an unknown class. How do we accomplish this?
    $\Rightarrow$ to store a reference to a pay-off object instead of a pay-off object using pointers or references.

A First Solution

Vanilla1.h

```
#ifndef __Option_Class__Vanilla1___
#define __Option_Class__Vanilla1___

#include <iostream>
#include "PayOff2.h" // use payoff class


// want the vanilla option to store its own copy of payoff object, not just a r
class VanillaOption
{
public:
    VanillaOption(PayOff& ThePayOff_, double Expiry_);
    double GetExpiry() const; // giving the expiry of the option
    double OptionPayOff(double Spot) const; // return the payoff at expiry give

private:
    double Expiry;
    PayOff& ThePayOff;
};
```

A First Solution

SimpleMonteCarlo3.h

```
#ifndef __Option_Class__SimpleMonteCarlo3__
#define __Option_Class__SimpleMonteCarlo3__

#include <iostream>
#include "Vanilla1.h"

double SimpleMonteCarlo3(const VanillaOption& TheOption,
                         double Spot,
                         double Vol,
                         double r,
                         unsigned long NumberOfPaths);

#endif /* defined(__Option_Class__SimpleMonteCarlo3__) */
```

A First Solution

- Let's look at this new project with the VanillaOption class:
  - DoubleDigital.h
  - DoubleDigital.cpp
  - PayOff2.h
  - PayOff2.cpp
  - Vanilla1.h
  - Vanilla1.cpp
  - SimpleMonteCarlo3.h
  - SimpleMonteCarlo3.cpp
  - VanillaMain1.cpp
- This program will compile and run but I do not like it. Why?
  - The vanilla option will not exist as independent object in its own, but will be always dependent on the PayOff object constructed outside the class.
  - If the PayOff object had been created using new as we did in the last chapter then it might be deleted before the option creased to exist which would result in the vanilla option calling methods of a non-existent object.

Breakout Exercise: New Method

- Test how fast *new* is on your computer and compiler. Do this by using it to allocate an array of new class called DoubleItem. The DoubleItem class takes a double value and stores it. And then create ten thousand Objects. See how the speed varies with array size.
- If you have more than one compiler see how they compare. Not that you can time things using the *clock*() function.
  - You will work as a team to write the code and to complete the design.
  - Run your code 100 times, 10000 times, and then 100000 times; compare the three times to take complete the execution.
  - Create plot using Excel to see whether it is a linear line where x is the size of the array, and y is the time to complete the execution.

The Rule of Three

- What do we really want to do? We want the vanilla option to store its own copy of the pay-off. However, we do not want the vanilla option to know the type of the pay-off object nor anything about any of its inherited classes for all the reasons we discussed previously. Our solution is to use virtual functions:
  - The object knows itself so it can make a copy of itself.
  - Thus, we define a virtual method of the base class which causes the object to create a copy of itself and return a pointer to the copy.
- Such a method is called a *virtual copy constructor*. The method is generally given the name clone.

```
virtual PayOff* clone() const = 0;

PayOff* PayOffCall::clone() const
{
    return new PayOffCall(*this);
}
```

The Rule of Three

- If we do not declare the copy constructor then the compiler will perform a *shallow copy* as oppose to a *deep copy*. When one of the two VanillaOption objects goes out of scope then its destructor will be called and the pointed-to PayOff object will be deleted. Thus if we write a class involving a destructor, we will generally need a copy constructor too.
- In fact, we have similar issues with the assignment operator so it is necessary to write it as well. This hence leads to the **'Rule of Three'** which states that
  - If any one of destructor, assignment operator and copy constructor is needed for a class then all three are.
  - Note for the copy constructor, the data members of the original and its copy will not be equal, since the pointers will have different values.
  - Note also that if we define a comparison operator (i.e. *operator* $==$) for the class, we would have to be careful to compare the objects pointed to rather than the pointers, or we would always get *false*.

The Bridge

- We have achieved what we desired: the vanilla option class can be happily copied and moved around just like any other class and it uses the code already written for the PayOff class so we have no unnecessary duplication.
- However, we have to write special code to handle assignment, construction and destruction. Every time we want to write a class with these properties, we will have to do the same thing again and again. There are two solutions:
  - Use a wrapper class that has been templatized: this is really generic programming rather than object-oriented programming.
  - Another is *the bridge pattern* that is a class does nothing except stores a pointer to an option pay-off and takes care of memory management.

The Bridge

PayOffBridge.h

```
#include <iostream>
#include "PayOff3.h"

class PayOffBridge
{
public:
    PayOffBridge(const PayOffBridge& original); // copy constructor
    PayOffBridge(const PayOff& innerPayOff);

    inline double operator()(double Spot) const;
    ~PayOffBridge();
    PayOffBridge& operator=(const PayOffBridge& original);

private:
    PayOff* ThePayOffPtr;
};

inline double PayOffBridge::operator()(double Spot) const
{
    return ThePayOffPtr -> operator()(Spot); // call the method in PayOff class
}
```

The Bridge

- Let’s look at this new project with the VanillaOption class:
    - DoubleDigital.h
    - DoubleDigital.cpp
    - PayOffBridge.h
    - PayOffBridge.cpp
    - PayOff3.h
    - PayOff3.cpp
    - Vanilla3.h
    - Vanilla3.cpp
    - VanillaMain2.cpp
- Everything in *Vanilla*3.*h* is totally straightforward as all the work has already been done by the bridge class
- The compiler automatically accepts the inherited class object as a substitute for the base class object, and then silently converts it for us into the *PayoffBridge* object which is then passed to the *VanillaOption* constructor.

The Parameter Class

- In practice, we want to generalize the parameters for our Monte Carlo routine to accommodate other types of options where more parameters are required. We naturally come to a *parameters* class. What should a parameters class do?
    - We want it to be able to store parameters such as volatility or interest rates or in a more general set-up, jump intensity.
- What information will we want to get out from a parameters class?
    - When implementing a financial model, we never actually need the instantaneous value of parameter: it is always the integral or the integral of the square that is important.
- What kind of differing parameters might we want?
    - It is important in object-oriented programming to think to the future, and therefore think about not just what we want now but what we might want in the future. Some examples would be a polynomial, a piece-wise constant function, and an exponentially decaying function. For all of these, finding the integral and square integral over definite intervals is straightforward.

The Parameter Class

- Let's look at this new project with the VanillaOption class:
  - DoubleDigital.h
  - DoubleDigital.cpp
  - PayOffBridge.h
  - PayOffBridge.cpp
  - PayOff3.h
  - PayOff3.cpp
  - Parameters.h
  - Parameters.cpp
  - VanillaMain4.cpp
- Note that by forcing us to go via the methods of *Parameters* class, the new design makes the code more comprehensible. We know immediately from looking at it that the integral of r is used, as is the square-integral of the volatility.

Key Points

- Cloning gives us a method of implementing a virtual copy constructor.
- The rule of three says that if we need any one of copy constructor, destructor, and assignment operator then we need all three.
- We can use a wrapper class to hide all the memory handling, allowing us to treat a polymorphic object just like any other object.
- The bridge pattern allows us to separate interface and implementation, enabling us to vary the two independently.
- We have to be careful to ensure that self-assignment does not cause crashes. The *new* command is slow.