Notes on C++ Design Patterns Derivatives Pricing

2021 – 04 – 23

Junfan Zhu

(junfanz@gatech.edu; junfanzhu@uchicago.edu)

Book Links

C++ Design Patterns and Derivatives Pricing (Mathematics, Finance and Risk, Series Number 2) 2nd Edition, by M. S. Joshi

> https://www.amazon.com/Patterns-Derivatives-Pricing-Mathematics-Finance/dp/0521721628

Table of Contents

- 4.5. Parameter Class
- 4.6. Summary
- 5. Strategies, decoration and statistics
- 5.1. Strategy Pattern
- 5.2. Designing a stats gatherer
- 5.3. Templates and Wrappers
- 5.4. Convergence table
- 5.5. Decoration
- 5.6. Summary
- 6. Random Numbers Class
- 6.1. Linear Congruential Generator & Aadpter Pattern
- 6.2. Anti-thetic sampling via decoration
- 6.3. Summary
- 7. Exotics engine and template pattern
- 7.1. Intro
- 7.2. Identifying components
- 7.3. Communication between components
- 7.4. Base Classes
- 7.5. Black-Scholes path generation engine
- 7.6. Arithmetic Asian Option
- 7.7. Summary
- 8. Trees
- 8.1. Intro
- 8.2. `TreeProduct` class
- 8.3. Tree class
- 8.4. Pricing on the tree
- 8.5. Summary
- 9. Solvers, templates, implied vol
- 9.1. Function objects
- 9.2. Newton-Raphson and funtion template arguments
- 9.3. Using Newton-Raphson to do implied vol
- 9.4. Pros and Cons of templation
- 9.5. Summary
- 10. Factory
- 10.1. Intro
- 10.2. Singleton Pattern
- 10.3. Factory coding
- 10.4. Automatic registration
- 10.5. Summary
- 11. Design Pattern revisited
- 11.1. Creational patterns
  - 11.1.1. Virtual copy constructor

---

1. Simple Monte Carlo

```
#include <Random1.h>
#include <iostream>
#include <cmath>
using namespace std;
```

```
double SimpleMonteCarlo1(double Expiry,
                         double Strike,
                         double Spot,
                         double Vol,
                         double r,
                         unsigned long NumberOfPaths)
{
    double variance = Vol * Vol * Expiry;
    double rootVariance = sqrt(variance);
    double itoCorrection = -0.5 * variance;

    double movedSpot = Spot*exp(r*Expiry + itoCorrection);
    double thisSpot;
    double runningSum = 0;

    for (unsignd long i=0; i<NumberOfPaths; i++)
    {
        double thisGaussian = GetOneGaussianByBoxMuller();
        thisSpot = movedSpot*exp(rootVariance * thisGaussian);
        double thisPayoff = thisSpot - Strike;
        thisPayoff = thisPayoff > 0 ? thisPayoff : 0;
        runningSum += thisPayoff;
    }
    double mean = runningSum / NumberOfPaths;
    mean *= exp(-r * Expiry);
    return mean;
}
int main()
{
    double Expiry;
    double Strike;
    double Spot;
    double Vol;
    double r;
    unsigned long NumberOfPaths;
    cout << "\nEnter expiry\n";
    cin >> Expiry;
    cout << "\nEnter strike\n";
    cin >> Strike;
    cout << "\nEnter spot\n";
    cin >> Spot;
    cout < "\nEnter vol\n";
    cin >> Vol;
    cout << "\nr\n";
    cin >> r;
```

```
    cout << "\nNumber of Paths\n";
    cin >> NumberOfPaths;

    double result = SimpleMonteCarlo1(Expiry,
        Strike,
        Spot,
        Vol,
        r,
        NumberOfPaths);
    cout << "the price is" << result << "\n";
    double tmp;
    cin >> tmp;
return 0;

// Random1.h

#ifndef RANDOM1_H
#define RANDOM1_H
double GetOneGaussianBySummation();
double GetOneGaussianByBoxMuller();
#endif
}

// Random1.cpp

#include <Random1.h>
#include <cstdlib>
#include <cmath>

#if !defined(_MSC_VER)
using namespace std;
#endif

double GetOneGaussianBySummation()
{
    double result = 0;
    for (unsigned long j =0; j<12; j++)
        result += rand()/static_cast<double>(RAND_MAX);
        result -= 6.0;
    return results;
}
double GetOneGaussianByBoxMuller()
{
    double result;
    double x;
    double y;
```

```
    double sizeSquared'
    do
    {
        x = 2.0 * rand()/static_cast<double>(RAND_MAX)-1;
        y = 2.0 * rand()/static_cast<double>(RAND_MAX)-1;
        sizeSquared = x*x + y*y;
    }
    while (sizeSquared >= 1.0);
    resullt = x*sqrt(-2*log(sizeSquared)/sizeSquared);
    return result;
}
```

1.1. Issues: Random number generation

Relies on the inbuilt generator which we don't know. We also want flexibility of using low-discrepancy numbers which means another form of generation. (Box-Muller doesn't work well with low discrepancy numbers, so we need flexibility in the inputs.)

2. Encapsulation

```
// PayOff1.h
#ifndef PAYOFF_H
#define PAYOFF_H

class PayOff
{
public:
    enum OptionType {call, put};
    PayOff(double Strike_, OptionType TheOptionsType_); // constructor
    double operator()(double Spot) const;
    // main method of the class
private:
    double Strike;
    OptionType TheOptionsType;
};
#endif

// PayOff1.cpp
#include <PayOff1.h>
#include <MinMax.h>

PayOff::PayOff(double Strike_, OptionType TheOptionsType_):
```

```
    Strike(Strike_), TheOptionsType(TheOptionsType_)
    {}

double PayOff::operator() (double spot) const
{
    switch (TheOptionsType)
    {
        case call:
            return max(spot-Strike,0.0);
        case put:
            return max(Strike-spot, 0.0);
        default:
            throw("unknown option type found.");
    }
}

// SimpleMC.h
#ifndef SIMPLEMC_H
#define SIMPLEMC_H
#include <PayOff1.h>

double SimpleMonteCarlo2(const PayOff& thePayOff,
                         double Expiry,
                         double Spot,
                         double Vol,
                         double r,
                         unsigned long NumberOfPaths);
#endif

// SimpleMC.cpp
#include <SimpleMC.h>
#include <Random1.h>
#include <cmath>
#if !defined(_MSC_VER)
using namespace std;
#endif

double SimpleMonteCarlo2(const PayOff& thePayOff,
                         double Expiry,
                         double Spot,
                         double Vol,
                         double r,
                         unsigned long NumberOfPaths)
{
    double variance = Vol * Vol * Expiry;
    double rootVariance = sqrt(variance):
```

```
    double itoCorrection = -0.5*variance;
    double movedSpot = Spot * exp(r*Expiry + itoCorrection);
    double thisSpot;
    double runningSum = 0;
    for (unsigned long i=0; i < NumberOfPaths; i++)
    {
        double thisGaussian = GetOneGaussianByBoxMuller();
        thisSpot = movedSpot * exp(rootVariance * thisGaussian);
        double thisPayOff = thePayOff(thisSpot);
        runningSum += thisPayOff;
    }
    double mean = runningSum / NumberOfPaths;
    mean *= exp(-r * Expiry);
    return mean;
}

// SimpleMCMain2.cpp
#include <SimpleMC.h>
#include <iostream>
using namespace std;

int main()
{
    double Expiry;
    double Strike;
    double Spot;
    double Vol;
    double r;
    unsigned long NumberOfPaths;
    cout << "\nEnter expiry\n";
    cin >> Expiry;
    cout << "\nEnter Strike\n";
    cin >> Strike;
    cout << "\nEnter spot\n";
    cin >> Spot;
    cout << "\nEnter vol\n";
    cin >> Vol;
    cout << "\nr\n";
    cin >> r;
    cout << "\nNumber of paths\n";
    cin >> NumberOfPaths;
    PayOff callPayOff(Strike, PayOff::call);
    PayOff putPayOff(Strike, PayOff::put);
    double resultCall = SimpleMonteCarlo2(callPayOff, Expiry, Spot, Vol, r, NumberOfPaths);
    double resultPut = SimpleMonteCarlo2(putPayOff, Expiry, Spot, Vol, r, NumberOfPaths);
    cout << " the prices are " << resultCall<< "for the call and "<< resultPut << "for the p
```

```
    double tmp;
    cin >> tmp;
    return 0;
}
```

- `enum` to distinguish between different sorts of pay-offs. If we want more than put and call, we would add them to the list.
- We have overloaded `operator()`. To call this method for an object `thePayoff` with spot given by S, we can say `thePayoff(S)`. `operator()` is const, so that the method doesn't modify the object, computing the pay-off doesn't change the strike or type of an option.

2.1. Open-closed principle

- **Open**: code should be open for extension
- **Closed**: File is closed for modification. We should do extension without modifying any existing code, and changing existing files.

Ex

Instead of making the class contain an `enum` that defines the option type, we use a function pointer. We replace `OptionType` with `double (*FunctionPtr)(double, double)`. The constructor for the pay-off would then take in the strike and a function pointer. When `operator()` was called, it would dereference the pointer and call the function pointed to with spot and strike as its arguments.

We can put the function pointed to in a new file, so the existing file for the pay-off class doesn't change each time we add a new form of pay-off. Neither the pay-off file nor the Monte Carlo file which includes the header file for payoff need to be recompiled.

3. Inheritance and virtual functions

```
#ifndef PAYOFF2_H
#define PAYOFF2_H
class PayOff
{
public:
    PayOff(){};
    virtual double operator()(double Spot) const = 0;
    virtual ~PayOff(){}
private:
};
```

```
class PayOffCall : public PayOff // inheritance
{
public:
    PayOffCall(double Strike_);
    virtual double operator()(double Spot) const;
    virtual ~PayOffCall(){}
private:
    double Strike;
};
class PayOffPut : public PayOff
{
public:
    PayOffPut(double Strike_);
    virtual double operator()(double Spot) const;
    virtual ~PayOffPut(){}
private:
    double Strile;
};
#endif

// PayOff2.cpp
#include <PayOff2.h>
#include <minmax.h>
PayOffCall::PayOffCall(double Strike_) : Strike(Strike_)
{}
double PayOffCall::operator() (double Spot) const
{
    return max(Spot - Strike, 0.0);
}
double PayOffPut::operator() (double Spot) const
{
    return max(Strike - Spot, 0.0);
}
PayOffPut::PayOffPut(double Strike_) : Strike(Strike_)
{}
```

3.1. Virtual Functions

- Adding keyword *virtual* to `operator()` at Pay-off class, and add `=0` at the end of `operator()`. Add destructor which is also `virtual`.
- `opaerator()` is *virtual* function, its address is bound at runtime instead of at compile time. Where `PayOff` object has been specified in simple Monte Carlo, the code will encounter an object of a class that has been inherited from `PayOff`. It will then decide what function to call on the basis of what type that object is. If the object is of type `PayOffCall`, it calls the

`operator()` method of `PayOffCall`, and if it is of type `PayOffPut`, it uses the method from the `PayOffPut` class.

- How compiler implements virtual function? It adds extra data to each object of the class, which specifies what function to use. What virtual function stores is a function pointer. The virtual function table is a list of addresses to be called for the virtual functins associated with the class. This operation takes a small amount of time, and amount of memory per object has increased as the object contains extra data.
- Virtual function `operator()` has `=0` after it, so it's *pure* virtual function, which doesn't need to be defined in the base class, and must be defined in inherited class. `=0` means the class is incompllete, and certain aspects of interface must be programmed in an inherited class.

3.1.1. Why virtual function

- Virtual function is simple in syntactic, and structure of program is clear.
- We get extra functionality. With inheritance, we simply require the inherited class to cntain all data necessary, if we want to do a complicated pay-off as a linear combination of call options, the extra data could be further call options whose payoffs would be evaluated inside the `operator()` method and added together.

3.2. Pass the inherited object by reference

- Parameter `thePayOff` has type `const PayOff&`, as the parameter is passed by reference rather than by value by `&`. If no `const`, the code won't compile, because compiler refuses to create an object of type `PayOff` as it has a pure virtual method, and method `operator()` has not been defined. So you can't create an object of a type with a pure virtual method.
- When compiler encounters the argument of type `PayOff`, the input parameter is copied into a variable oftype `PayOff`, because the compiler will call the copy constructor of `PayOff`. Copy constructor takes in an object of type `const PayOff&`.
- Disastrous things would occur if the new object inherited the virtual function table of the inherited object, as the virtual methods would access non-existent data members with dangerous consequences. Making the base class method concrete instead of pure virtual is a mistake.
- What happens when object is passed by reference? The function is passed the address of the object in memory, no copying ccurs. We include `const` to indicate that the routine can't change the state of the object. The function can look but not touch.
- `new PayOffCall(Strike)` creates `PayOffCall` object and returns a pointer of type `PayOffCall*` to the object for us to access. We can cast a pointer from an inherited class pointer into a base class pointer. `new` =

compiler must not destrooy the object nor deallocate the momery unless we say so. We need to `delete`. The base class is abstract and calling the base class destructor must be an error, so we declare the destructor `virtual`. The compiler use the virtual function table to decide which destructor to call.

```
#ifndef DOUBLEDIGITAL_H
#define DOUBLEDIGITAL_H
#include <Payoff2.h>

class PayOffDoubleDigital : public PayOff
{
public:
    PayOOffDoubleDigital(double LowerLevel_, double UpperLevel_);
    virtual double operator()(double Spot) const;
    virtual ~PayOffDoubleDigital(){}
private:
    double LowerLevel;
    double UpperLevel;
};
#endif

// DoubleDigital.cpp
#include <DoubleDigital.cpp>
#include <DoubleDigital.h>
PayOffDoubleDigital::PayOffDoubleDigital(double LowerLevel_, double UpperLevel_)
    : LowerLevel(LowerLevel_), UpperLevel(UpperLevel_)
{}
double PayOffDoubleDigital::operator()(double Spot) const
{
    if (Spot <= LowerLevel)
        return 0;
    if (Spot >= UpperLevel)
        return 0;
    return 1;
}
```

We use inheritance, `PayOff` class is open for extension, but closed for modification.

3.3. Summary

- Virtual function is bouond at run time rather than at compile time.
- We can't have objects from classes with pure virtual functions.
- We should pass inherited class objects by reference if we don't wish to change the virtual functions.

- Virtual functions are implemented via a table of function pointers.
- If a class has a pure virtual functons, then it should have a virtual destructor.

4. Virtual Constructor

4.1. Problem

We want a vanilla option object to be able to contain an object from an unknown class, and we need to store a reference to a payoff pbject. But if the `VanillaOption` class stores a reference to a `PayOff` object which is defined outside the class, then if we change that object, then the pay-off of the vanilla option will change. The vanilla optin won't exist as independent object on its wn, but will always be dependent n the `PayOff` object constructed outside class. Moreover, if the `PayOff` object is created using `new`, then it might be `deleted` before the option ceased to exis, which would result in vanilla option calling methods of a non-existent object.

4.2. Virtual Construction

We want vanilla option to store its own copy of the payooff. But we don't want vanilla option to know the type of payoff object or its inherited class.So 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. (*virtual copy constructor* named `clone`)

```
// add pure virtual method to base class
virtual PayOff* clone() const=0;
// define it in each inherited class
PayOff* PayOffCall::clone() coonst
{
    // create a copy of the object, for which clone method is called
    // this pointer points to the object being called
    // call to clone  = call to copy constructor of PayffCall, ..
    // .. which returns a copy of the original PayOffCall
    // operator new is used, so the object will continue to exist
    return new PayOffCall(*this);
}
```

Modified code

```
#ifndef PAYOFF3_H
#define PAYOFF3_H
```

```
class PayOff
{
public:
    PayOff(){};
    virtual double operator()(double Spot) const=0;
    virtual ~PayOff(){}
    virtual PayOff* clone() const = 0;
private:
};

class PayOffCall : public PayOff
{
public:
    PayOffCall(double Strike_);
    virtual double operator()(double Spot) const;
    virtual ~PayOffCall(){}
    virtual PayOff* clone() const;
private:
    double Strike;
};

class PayOffPut : public PayOff
{
public:
    PayOffPut(double Strike_);
    virtual double operator()(double Spot) const;
    virtual ~PayOffPut(){}
    virtual PayOff* clone() const;
private:
    double Strike;
};
#endif

// PayOff3.cpp
#include <PayOff3.h>
#include <minmax.h>
PayOffCall::PayOffCall(double Strike_) : Strike(Strike_)
{}
double PayOffCall::operator() (double Spot) coonst
{
    return max(Spot - Strike, 0.0);
}
PayOff * PayOffCall::clone() const
{
    return new PayOffCall(*this);
}
```

```
double PayOffPut::operator() (double Spot) const
{
    return max(Strike-Spot, 0.0);
}
PayOffPut::PayOffPut(double Strike_): Strike(Strike_)
{}
PayOff* PayOffPut::clone() const
{
    return new PayOffPut(*this);
}

// Vanilla2.h
#ifndef VANILLA_2_H
#define VANILLA_@_H
#include <PayOff3.h>
class VanillaOption
{
public:
    VanillaOption(const PayOff& ThePayOff_, double Expiry_);
    VanillaOption(const VanillaOption& original);
    VanillaOption& operator=(const VanillaOption& original);
    ~VanillaOption();
    double GetExpiry() const;
    double OptionPayOff(double Spot) const;
private:
    double Expiry;
    PayOff* ThePayOffPtr;
};
#endif

// Vanilla2.cpp
#include <Vanilla2.h>
VanillaOption::VanillaOption(const PayOff&
                             ThePayOff_, double Expiry_)
                             : Expiry(Expiry_)
    {
        The PayOffPtr = ThePayOff_.clone()
    }
double VanillaOption::GetExpiry() const
{
    return Expiry;
}
double VanillaOption::OptionPayOff(double Spot) const
{
    return (*ThePayOffPtr)(Spot);
}
```

```
VanillaOption::VanillaOption(const VanillaOption& original)
{
    Expiry = original.Expiry;
    ThePayOffPtr = original.ThePayOffPtr->clone();
}
VanillaOption& VanillaOption::
                operator=(const VanillaOption& original)
{
    if (this != &original)
    {
        Expiry = original.Expiry;
        delete ThePayOffPtr;
        ThePayOffPtr = original.ThePayOffPtr->clone();
    }
    return *this;
}
VanillaOption::~VanillaOption()
{
    delete ThePayOffPtr;
}
```

- Differences between `VanillaOption` and other classes: we include an assignment operator `operator=` and a copy constructor. If we don't declare copy constructor, then compiler will *shallow copy*[1]. 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, and if you use its pointer, it will lead to non-existent object being called.
- So, if we write a class with destructor, we will need a copy constructor too. ***Rule of Three***: if any one of destrucot, assignment operator, and copy constructor is needed for a class, then all three are.
- Copy constructor of `VanillaOption`. It copies the value of `Expiry` from original into the copy. We call `clone` method to copy the `PayOff` object stored in the copy's pointer. And the original and its copy aren't equal, since the poointers have different values, and yes we don't want them to use the original `PayOff` object. If we want to define a comparison operator `operator==` for the class, we should compare the objects pointed to rather than the pointers.

4.3. Bridge Pattern

Because we had to write special code too handle assignment, construction and destruction, every time. We can use a wrapper class that is templatized (generic

[1]Shallow copy: data members are copied, but no memory allocation. Any modification to objects we have pointed-to, will have the same effect in each copy.

programming); or by bridge pattern, we take vanilla option class and get rid of member `Expiry` and method `GetExpiry`. Then we only has a class to store a pointer to an option payoff and takes care of memory handling.

```
// PayOffBridge.h
#ifndef PAYFFBRIDGE_H
#define PAYOFFBRIDGE_H
#include <PayOff3.h>
class PayOffBridge
{
public:
    PayOffBridge(cnst PayOffBridge& original);
    PayOffBridge(const PayOOff& 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);
}
#endif

// PayOffBridge.cpp
#include<PayOffBridge.h>
PayOffBridge::PayOffBridge(const PayOffBridge& original)
{
    ThePayffPtr = original.ThePayOffPtr->clone();
}
PayOffBridge::PayOffBridge(const PayOff& innerPayOff)
{
    The PayOffPtr = innerPayOff.clone();
}
PayOffBridge& PayffBridge::operator = (const PayOffBridge& original)
{
    if (this != &original)
    {
        delete ThePayOffPtr;
        ThePayOffPtr = original.ThePayOffPtr->clone();
    }
    return *this;
}
// Vanilla3.h
#ifndef VANILLA_3_H
```

```
#define VANILLA_3_H
#include <PayOffBridge.h>
class VanillaOption
{
public:
    VanillaOption(const PayOffBridge& ThePayOff_, double Expiry);
    double OptionPayOOff(duoble Spot) const;
    double GetExpiry() const;
private:
    double Expiry;
    PayOffBridge ThePayOff;
};
#endif
```

4.4. new

`new` is slow. Every time we copy a bridged object, we implicitly call the `new`. The compiler has an area of memory as a stack, but when we use `new`, we want to destroy variables in *different* order, and since the variable continue to exist, so we need to use the stack to scan down the variable, and move all variables up the stack down to cover the gap. So it's time costing. This way, compiler doesn't use stack, but heap instead. Everytime `new` is called, the compiler find an empty piece of memory which is large and mark the memory as being in use. When `delete` is called, the memory is free.

4.5. Parameter Class

To store parameters such as volatility, interest rate, jump intensity, etc. When implementing a financial model, we don't need instantaneoous value of parameter, it's always the integral of the integral of the square that's important.

We employ the bridge design, define a class `Parameters` that handles the interactin without the outside world, and memory handling. Its only data member is a pointer to an abstract class `ParametersInner`, which defines the interface that the concrete classes we implement must fulfill. We also have `clone` method, `Integral` and `IntegralSquare` methods, so that wrapper class `Parameters` can copy.

```
// Parameters.h
#ifndef PARAMETERS_H
#define PARAMETERS_H
class ParametersInner
{
public:
```

```
    ParametersInner(){}
        virtual ParametersInner* clone() const=0;
        virtual double Integral(double time1, double time2) const = 0;
        virtual double IntegralSquare(double time1, double time2) const = 0;
        virtual ~ParametersInner() {}
private:
};
class Parameters
{
public:
    Parameters(const ParametersInner& innerObject);
    Parameters(const Parameters& original);
    Parameters& operator=(const Parameters& original);
    virtual ~Parameters();
    inline double Integral(double time1, double time2) const;
    inline double IntegralSquare(double time1, double time2) const;
    double Mean(double time1, double time2) const;
    double RootMeanSquare(double time1, double time2) const;
private:
    ParametersInner* InnerObjectPtr;
};
inline double Parameters::Integral(double time1, double time2) const
{
    return InnerObjectPtr->Integral(time1, time2);
}
inline double Parameters::IntegralSquare(double time1, double time2) const
{
    return InnerObjectPtr->IntegralSquare(time1, time2);
}
class ParametersConstant : public ParametersInner
{
public:
    ParametersConstant(double constant);
    virtual ParametersInner* clone() const;
    virtual double Integral(double time1, double time2) const;
    virtual double IntegralSquare(double time1, double time2) const;
private:
    double Constant;
    double ConstantSquare;
};
#endif

// Parameters.cpp
#include <Parameters.h>
Parameters::Parameters(const ParametersInner& innerObject)
{
```

```
    InnerObjectPtr = innerObject.clone();
}
Parameters::Parameters(const Parameters& original)
{
    InnerObjectPtr = original.InnerObjectPtr->clone();
}
Parameters& Parameters::operator = (const Parameters& original)
{
    if (this != &original)
    {
        delete InnerObjectPtr;
        InnerObjectPtr = original.InnerObjectPtr->clone();
    }
    return *this;
}
Parameters::~Parameters()
{
    delete InnerObjectPtr;
}
double Parameters::Mean(double time1, double time2) const
{
    double total = Integral(time1, time2);
    return total/(time2-time1);
}
double Parameters::RootMeanSquare(double time1, double time2) const
{
    double total = IntegralSquare(time1, time2);
    return total/(time2-time1);
}
ParametersConstant::ParametersConstant(double constant)
{
    Constant = constant;
    ConstantSquare = Constant * Constant;
}
ParametersInner* ParametersConstant::clone() const
{
    return new ParametersConstant(*this);
}
double ParametersConstant::Integral(double time1, double time2) const
{
    return (time2-time1)*Constant;
}
double ParametersConstant::IntegralSquare(double time1, double time2) const
{
    return (time2-time1)*ConstantSquare;
}
```

```
// SimpleMC6.h
#ifndef SIMPLEMC6
```

4.6. Summary

- Cloning can implement virtual copy constructor.
- The rule of three: 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.
- Bridge pattern can separate interface and implementation, so as to vary the two independently.
- `new` is slow.
- Be careful to ensure that self-assignment doesn't cause crashes.

5. Strategies, decoration and statistics

5.1. Strategy Pattern

Strategy pattern is to use an auxiliary class to decide how part of an algorithm is implemented. We develop an object-oriented routine, make statistics gatherer an input, and Monte Carlo routine will take in a statistics gatherer object, store results in it, and statistics gatherer will output the statistics. The statistics gatherer is reusable.

5.2. Designing a stats gatherer

We design a statistics to be called many times and more robust than object that creashes if get results method called twice. We make it a const method as it won't change the state of the object.

```
// MCStatistics.h
#ifndef STATISTICS_H
#define STATISTICS_H
#include <vector>
class StatisticsMC
{
public:
    StatisticsMC(){}
    virtual void DumpOneResult(double result) = 0;
    virtual std::vector<std::vector<double>>
```

```
            GetResultsSoFar() const = 0;
    virtual StatisticsMC* clone() const = 0;
    virtual ~StatisticsMC(){}
private:
};

class StatisticsMean: public StatisticsMC
{
public:
    StatisticsMean();
    virtual void DumpOneResult(double result);
    virtual std::vector<std::vector<double>>
            GetResultsSoFar() const;
    virtual StatisticsMC* clone() const;
private:
    double RunningSum;
    unsigned long PathsDoone;
};
#endif
```

Our abstract base class is `StatisticsMC`, it has pure virtual functions `DumpOneResult` and `GetResultsSoFar`. We use `clone` methdod to allow for virtual copy construction. We make destructor virtual, as any cloned object will be destroyed via pointers to the base class which won't know their type. The base class defines interface.

```
// MCStatistics.cpp
#include<MCStatistics.h>
using namespace std;
StatisticsMean :: StatisticsMean()
    :
    RunningSum(0.0), PathsDone(OUL)
    {}
void StatisticsMean::DumpOneResult(double result)
{
    PathsDone++;
    RunningSum += result;
}
vector<vector<double>> StatisticsMean::GetResultsSoFar() const
{
    vector<vector<double>> Results(1);
    Results[0].resize(1);
    Results[0][0] = RunningSum / PathsDonel;
    return Results;
}
```

```
StatisticsMC* StatisticsMean::clone() const
{
    return new StatisticsMean(*this);
}
```

5.3. Templates and Wrappers

We have to manually copy objects, and we want to write a wrapper class for it. Wrapper can act like a pointer to single object but with added responsibilities: pointer is both responsible for and owns the object pointed to. If we copy `Wrapper` object ,the pointed-to object is also copies, so that each `Wrapper` object has its own copy of pointed-to object. When Wrapper ceases to exist because of going out of scope or deleted, the pointed-to-object is automatically deleted as well.

```
// Wrapper.h

#ifndef WRAPPER_H
#define WRAPPER_H
template <class T>
class Wrapper
{
public:
    Wrapper()
    { DataPtr = 0;}
    Wrapper(const T& innner)
    {
        DataPtr = inner.clone();
    }
    ~Wrapper()
    {
        if (DataPtr != 0)
            delete DataPtr;
    }
    Wrapper(const Wrapper<T>& original)
    {
        if (original.DataPtr != 0)
            DataPtr = originnal.DataPtr->clone();
        else
            DataPtr = 0;
    }
    Wrapper& operator= (const Wrapper<T>& original)
    {
        if (this != &original)
        {
```

```
            if (DataPtr!=0)
                delete DataPtr;
            DataPtr = (original.DataPtr != 0) ?
                original.DataPtr->clone() : 0;
        }
        return *this;
    }

    T& operator*()
    {
        return *DataPtr;
    }
    const T& operator*() const
    {
        return *DataPtr;
    }
    const T* const operator->() const
    {
        return DataPtr;
    }
    T* operator->()
    {
        return DataPtr;
    }
private:
    T* DataPtr;
};
#endif
```

`Wrapper` object can point to nothing, so we need to set pointer to 0 at that time. When carrying oout copying and assignment, we have to take care of special case. For const version, return const object; for non-const, return non-const. So we declare 2 operators `inline` to ensure there's no performance overhead included by going via a wrapper. We declare `operator->` to have both const and non-const versions.

5.4. Convergence table

Rather than returning stats for entire simulation, we can return them for every power of 2 to see how numbers are varying. We define class `ConvergenceTable` inherited [2] from `MCStatistics`, and has wrapper of `MCStatistics` as a data member.

[2] From outside, it looks like any other stats-gatherer object. Difference on the inside is that we can make data member refer to any kind of stats gatherer we like, and so we have a convergence table for any stats.

```cpp
// ConvergenceTable.h

#ifndef CONVERGENCE_TABLE_H
#define CONVERGENCE_TABLE_H
#include <MCStatistics.h>
#include <wrapper.h>
class ConvergenceTable : public StatisticsMC
{
public:
    ConvergenceTable(const Wrapper<StatisticsMC> & Inner_);
    virtual StatisticsMC* clone() const;
    virtual void DumpOneResult(double result);
    virtual std::vector<std::vector<double>>
        GetResultsSoFar() const;
private:
    Wrapper<StatisticsMC> Inner;
    std::vector<std::vector<double>> ResultsSoFar;
    unsigned long StoppingPoint;
};
#endif

// ConvegenceTable.cpp
#include <ConvergenceTable.h>
ConvergenceTable::ConvergenceTable(const
            Wrapper<StatisticsMC>& Inner_)
            : Inner(Inner_)
{
    StoppingPoint = 2;
    PathsDone = 0;
}
StatisticsMC* ConvergenceTable::clone() const
{
    return new ConvergenceTable(*this);
}
void ConvegenceTable::DumpOneResult(double result)
{
    Inner->DumpOneResult(result);
    ++PathsDone;
    if (PathsDone == StoppingPoint)
    {
        StoppingPoint *= 2;
        std::vector<std::vector<double>>
            thisResult(Inner->GetResultsSoFar());
        for (unsigned long i =0; i < thisResult.size(); i++)
        {
            thisResult[i].push_back(PathsDone);
```

```
            ResultsSoFar.push_back(thisResullt[i]);
        }
    }
    return;
}

std::vector<std::vector<double>>
        ConvergenceTable::GetResultsSoFar() const
{
    std::vector<std::vector<double>> tmp(ResultsSoFar);
    if (PathsDone * 2 != StoppingPoint)
    {
        std::vector<std::vector<douoble>>
            thisResult(Inner->GetResultsSoFar());
        for (unsigned long i=0; i < thisResult.size(); i++)
        {
            thisResult[i].push_back(PathsDone);
            tmp.push_back(thisResult[i]);
        }
    }
    return tmp;
}
```

5.5. Decoration

Decorator pattern: add functionality to a class without changing interface. Any decoration which can be applied to original class can be applied to decorated class.

We can write a decorator class containing a vector of stats gatherers and passes gathered value to each one individually.

5.6. Summary

- Strategy pattern: makes routiens more flexible, by making part of an algorithm be implemented by an inputted class.
- Template the code to save time in rewritting for similar code and across many different classes
- If we want containers of polymorphic objects, we must use wrappers or pointers.
- Decoration can add functionality by placing a class around a class which has the same interface: the outer class is inherited from the same base class. Class can be decorated many times.

6. Random Numbers Class

We want to encapsulate random number generation, because we can't expect consistency across compilers. If we want to test code by running it on multiple platforms, we can expect to obtain different streams of random numbers and our Monte Carlo simulation will converge to the same number, but it's a lot weakeer than having every single random draw matching. So our code is harder to test. `rand` is not predictable.

An iterator is a generalization of a pointer and we can templatize the code t work off any iterator.

```
//Random2.cpp

#include <Random2.h>
#include <Normals.h>
#include <cstdlib>
#if !defined(_MSC_VER)
using namespace std;
#endif

void RandomBase::GetGaussians(MJArray& variates)
{
    GetUniforms(variates);
    for (unsigned long i=0; i<Dimensionality; i++)
    {
        double x = variates[i];
        variates[i] = InverseCumulativeNormal(x);
    }
}
void RandomBase::ResetDimensionality(unsigned long NewDimensionality)
{
    Dimensionality = NewDimensionality;
}
RandomBase::RandomBase(unsigned long Dimensionality_)
    : Dimensionality(Dimensionality_)
    {}
```

6.1. Linear Congruential Generator & Aadpter Pattern

We write a random number generator with 2 pieces: small inner class that develops a random generator that returns 1 interger every time when called; a larger class that turns output into a vector of uniforms in format desired.

```
//ParkMiller.h
#ifndef PARK_MILLER_H
```

```
#define PARK_MILLER_H
#include <Random2.h>
class ParkMiller
{
public:
    ParkMiller(long Seed = 1);
    long GetOneRandomInteger();
    void SetSeed(long Seed);
    static unsigned long Max();
    static unsigned long Min();
private:
    long Seed;
};
class RandomParkMiller: public RandomBase
{
public:
    RandomParkMiller(unsigned long Dimensionality,
                     unsigned long Seed = 1);
    virtual RandomBase* clone() const;
    virtual void GetUniforms(MJArray& variates);
    virtual void Skip(unsigned long numberOfPaths);
    virtual void SetSeed(unsigned long Seed);
    virtual void Reset();
    virtual void ResetDimensionality(unsigned long NewDimensionality);
private:
    ParkMiller InnerGenerator;
    unsigned long InitialSeed;
    double Reciprocal;
};
#endif
```

Inner class develops sequence of uncorrelated `long`s. Bigger class is inherite dfrom `RandomBase` with all methods it requires.

Adapter Pattern: random generator's interface is not what the rest of code expects. So we write a class around it, which adapts its interface into what we want. Whenever we use old code or import libraries, it's rare for interface to fit precisely with what we have been using, and the adapter pattern is necessary. It's an intermediary class which transforms onne interface into another. It's like a plug adapter.

So, the generator relies on modular arithmetic, if you repeatedly multiply a number by a large number, and take modulus w.r.t. another number, then the successive remainders are effectively ranndom.

```
// ParkMiller.cpp
```

```
#include <ParkMiller.h>
const long a = 16807;
const long m = 2147483647;
const long q = 127773;
const long r = 2836;
ParkMiller::ParkMiller(long Seed_) : Seed(Seed_)
{
    if (Seed = 0)
        Seed = 1;
}
void ParkMiller::SetSeed(long Seed_)
{
    Seed = Seed_;
    if (Seed == 0)
        Seed = 1;
}
unsigned long ParkMiller::Max()
{
    return m-1;
}
long ParkMiller::GetOneRandomInteger()
{
    long k;
    k = Seed/q;
    Seed = a*(Seed-k*q) - r*k;
    if (Seed  0)
        Seed += m;
    return Seed;
}
RandomParkMiller::RandomParkMiller(unsigned long Dimensionality,
                unsignedd long Seed)
                : RandomBase(Dimensionality),
                InnerGenerator(Seed),
                InitialSeed(Seed)
                {
                    Reciprocal = 1/(1.0+InnereGenerator.Max());
                }
RandomBase* RandomParkMiller::clone() const
{
    return new RandomParkMiller(*this);
}
void RandomParkMiller::GetUniforms(MJArray& variates)
{
    for (unsigned long j=0; j < GetDimensionality(); j++)
        InnerGenerator.GetOneRandomInteger() * Reciprocal;
}
```

```
void RandomParkMiller::Skip(unsigned long numberOfPaths)
{
    MJArray tmp(GetDimensionality());
    for (unsigned long j=0; j < numberOfPaths; j++)
        GetUniforms(tmp);
}
void RandomParkMiller::SetSeed(unsigned long Seed)
{
    InitialSeed = Seed;
    InnerGenerator.SetSeed(Seed);
}
void RandomParkMiller::Reset()
{
    InnerGenerator.SetSeed(InitialSeed);
}
void RandomParkMiller::ResetDimensionality(unsigned long NewDimensionality)
{
    RandomBase::ResetDimensionality(NewDimensionality);
    InnerGenerator.SetSeed(InitialSeed);
}
```

6.2. Anti-thetic sampling via decoration

This causes simulation t converge faster.

```
// AntiThetic.cpp

#include <AntiThetic.h>
AntiThetic::AntiThetic(const Wrapper<RandomBase>& innerGenerator)
        : RandomBase(*innerGenerator),
        InnerGenerator(innerGenerator)
        {
            InnerGenerator -> Reset();
            OddEven = true;
            NextVariates.resize(GetDimensionality());
        }
RandomBase* AntiThetic::clone() const
{
    return new AntiThetic(*this);
}
void AntiThetic::GetUniforms(MJArray& variates)
{
    if (OddEven)
    {
        InnerGenerator->GetUniforms(variates);
```

```
        for (unsigned long i=0; i<GetDimensionality(); i++)
            NextVariates[i] = 1.0-variates[i];
        OddEven = false;
    }
    else
    {
        variates = NextVariates;
        OddEven = true;
    }
}
void AntiThetic::SetSeed(unsigned long Seed)
{
    InnerGenerator->SetSeed(Seed);
    OddEven = true;
}
void AntiThetic::Skip(unsigned long numberOfPaths)
{
    if (numberOfPaths == 0)
        return;
    if (OddEven)
    {
        OddEven = false;
        numberOfPaths--;
    }
    InnerGenerator->Skip(numberOfPaths / 2);
    if (numberOfPaths % 2)
    {
        MJArray tmp(GetDimensionality());
        GetUniforms(tmp);
    }
}
void AntiThetic::ResetDimensionality(unsigned long NewDimensionality)
{
    RandomBase::ResetDimensionality(NewDimensionality);
    NextVariates.resize(NewDimensionality);
    InnerGenerator->ResetDimensionality(NewDimensionality);
}
void AntiThetic::Reset()
{
    InnerGenerator->Reset();
    OddEven = true;
}
```

6.3. Summary

- `rand` is implementation-dependent. Results from `rand()` aren't easily reproducible.
- We need to be sure that random generator is capable of dimentionality necessary for a simulation.
- Using random number class allows us to use decoration.
- The inverse cumulative normal function is the most robustt way to turn uniform variates from open interval $(0, 1)$ to Gaussian variaets.
- Using random number class makes it easier to plug in low-discrepancy numbers
- Anti-thetic sampling can be implemented via decoration.

7. Exotics engine and template pattern

7.1. Intro

We want to develop a flexible Monte Carlo pricer for exotic options which pay ooff att some future date according to the value of spot on a finite number of dates. Derivatives could pay cash-flows at more than 1 time. *Ex*: discrette barrier knock-out option could pay ordinary vanilla pay-off at time of expiry, and a rebate at time of knock-out otherwise.

7.2. Identifying components

For each path, we generate a discounted payoff which is then averaged over all paths to obtain a price. To generate payoff for a path, we need to know path of stock prices at relevant times, plug this path into payoff funcction of option to obtain cashflows, and discount the cashflows back to start to obtain price for that path. A path-dependent exotic option class will encapsulate info written in term-sheet for the option.

4 actions:

- generation of stock price path
- generaion of cashflwos given a stock price path
- discounting and summing of cashflows for a given path
- averaging of prices over all paths

Template design pattern = base class sets up a structure with methdos that control everything, in case it would be methods to run simulation and account for each path. Though thee main work of actually generating the path isn't

defined in the base class, but is instead called via a pure virtual function, which is defined in an inherited class. So, as templatized code, the code is set up more to define a structure than to do the real world which is coded elsewhere. The option we pursue is to make path generation a virtual method of the base class.

7.3. Communication between components

The product takes in a vecttor of spot values for its relevant times and split out cash-flows generated. There has to be a mechanism for product to tell the path generator for which times it needs the value of spot, and we need to decide how to define cash-flow object.

`GetLookAtTimees` method passes back an array of times that are relevant to payooff function of the product. We can make cashflow a pair of `doubles` which define the amount and time of cashflow, whose disadvantage is that if we have a complicated term structure of interest rates, the action of computing discount fator for cashflow time may be slow, and with product being allowed to pass back an arbitrary time, this discounting will have to be done on every path. Although we can cache already-computed times, there is still problem that searching cache foor already-computed times will take time.

Many products can only pay off at one time, whicch means it's better to pre-compute discount factor for that time. But we still need to knoow what time is, so we require product to have anther method `PossibleCachFlowTimes` which returns an array defining possible times. Engine will knw all possible times in advance, we can return cashflow as a pair: index + amount.

`CashFlows` method takes in array defining spot values, and returns cashflows. We alloow more thann 1 cach flow, so we must use container to pass them back with STL vector class. But it's time consuming as we have to create a new vector every time method is called and need memory allcationn. So we take argument of type `vector<CashFlow>$`.

The vector should be in correct size, but resizing needs memory allocation, and some implementations of STL destroy all objects in vector during resize, so every resize involves looping, and is slow when no memory allocation is necessary. So we can tell how big the vector has to be, and then each time the method is called, return an unsigned long saying how many cashflows have been generated. So we have 2 pure virtual methods:

```
virtual unsigned long MaxNumberOfCashFlows() coonst = 0;
```

```
virtual unsigned long CachFlows(const MJArray& SpotValues, std::vector<CashFlow>&
GeneratedFlows) const=0;
```

So the communication is:

1. Path generator asks product what times it needs spot for, and it passes back array

2. Accounting part of engine asks product what cash-flow times are possible, and passes back an array. The engine computes all possible discount factors.
3. Accounting part of engine asks product max number of cash flows it can generate, and sets up vector of that size
4. For each path, engine gets array of spot values from path generator
5. Array of spot values is passed into product, which passes back number of cash-flows and puts value into vector
6. Cash-flows are discoounted appropriately and summed, and total value passed int stats gatherer
7. After loopinng is done, final results obtained from stats gatherer

7.4. Base Classes

```
// PathDependent.h
#ifndef PATH_DEPENDENT_H
#define PATH_DEPENDENT_H
#include <Arrays.h>
#include <vector>

class CashFlow
{
public:
    double Amount;
    unsigned long TimeIndex;
    CashFlow(unsigned long TimeIndex_ = OUL,
            double Amount_ = 0.0)
            : TimeIndex(TimeIndex_),
            Amount(Amount_){};
};

class PathDependent
{
public:
    PathDependent(const MJArray& LookAtTimes);
    const MJArray& GetLookAtTimes() const;
    virtual unsigned long MaxNumberOfCashFlows() const = 0;
    virtual MJArray PossibleCashFlowTimes() const = 0;
    virtual unsigned long CashFlows(const MJArray& SpotValues,
            std::vector<CashFlow>& GeneratedFlows) const = 0;
    virtual PathDependent* clone() const = 0;
    virtual ~PathDependent(){}
private:
    MJArray LookAtTimes;
```

```
};
#endif

// PathDependent.cpp
#include<PathDependent.h>
PathDependet::PathDependent(const MJArray& LookAtTimes_)
        : LookAtTimes(LookAtTimes_)
        {}
const MJArray& PathDependent::GetLookAtTimes() const
{
    return LookAtTimes;
}

// ExoticEngine.h
#ifndef EXOTIC_ENGINE_H
#define EXOTIC_ENGINE_H
#include <wrapper.h>
#include <Parameters.h>
#include <PathDependent.h>
#include <MCStatistics.h>
#include <vector>
class ExoticEngine
{
public:
    ExoticEngine(const Wrapper<PathDependent>&
        The Prouct_, const Parameters& r_);
    virtual void GetOnePath(MJArray& SpotValues)=0;
    void DoSimulation(StatisticsMC& TheGatherer,
            unsigned long NumberOfPaths);
    virtual ~ExoticEngine(){}
    double DoOnePath(const MJArray& SpotValues) const;
private:
    Wrapper<PathDependdent> TheProduct;
    Parameters r;
    MJArray Discounts;
    mutable std::vector<CashFlow> TheseCashFlows;
};
#endif
```

We have array `Discounts`, which will be used to store discount factors in orde rfor the possible cash-flow times. We have mutable data member `TheseCashFlows`: it an change value inside const member functions. Variable is not only a data member, but instead it's a workspace variable: it's faster too declare once and for all in the class definition. We design the class and vector is created once and for all, so no need to cost time for destroying containers. We split out main method, it has **2 auxiliary methods**: `DoOnePath` and `GetOnePath`. The

second is pure virtual and can be defined in an inherited class which will involve a choice of stochastic process. Both methods pass arrays by reference to avoid memory allocation.

```
// ExoticEngine.cpp

#include <ExoticEngine.h>
#include <cmath>
ExoticEngine::ExoticEngine(const Wrapper<PathDependent>&
        TheProduct_, const Parameters& r_)
        :
        TheProduct(TheProuct_),
        r(r_),
        Discounts(TheProduct_->PossibleCashFlowTimes())
{
    for (unsigned long i=0; i < Discounts.size(); i++)
        Discounts[i] = exp(-r.Integral(0.0, Discounts[i]));
    TheseCashFlows.resize(TheProduct->MaxNumberOfCashFlows());
}
void ExoticEngine::DoSimulation(StatisticsMC& TheGatherer,
        unsigned long NumberOfPaths)
{
    MJArray SpotValues(TheProduct->GetLookAtTimes().size());
    TheseCashFlows.resize(TheProduct->MaxNumberOfCashFlows());
    double thisValue;
    for (unsigned long i=0; i<NumberOfPaths; ++i)
    {
        GetOnePath(SpotValues);
        thisValue = DoOnePath(SpotValues);
        TheGatherer.DumpOneResult(thisValue);
    }
    return;
}
double ExoticEngine::DoOnePath(const MJArray&
                SpotValues) const
{
    unsigned long NumberFlows =
            TheProduct->CashFlows(SpotValues, TheseCashFlows);
    double Value= 0.0;
    for (unsigned i=0; i < NumberFlows; ++i)
        Value += TheseCashFlows[i].Amount *
                Discounts[TheseCashFlows[i].TimeIndex];
    return Value;
}
```

Constructor stores inputs, computes discount facots and make sure cashflows vec-

tor is of correct size. `DoSimulation` loops through all paths, calling `GetOnePath` to get array of spot value and passes them into `DoOnePath` to get value for that set of spot values, which is dumped into stats gatherer.

7.5. Black-Scholes path generation engine

Stock price follows process $dS_t = (r(t) - d(t))S_t dt + \sigma(t)S_t dW_t$

To simulate the process, we set

$$\log S_{t_j} = logS_{t_{j-1}} + \int_{t_{j-1}}^{t_j} \left(r(s) - d(s) - \frac{1}{2}\sigma(s)^2 \right) ds + \sqrt{\int_{t_{j-1}}^{t_j} \sigma(s)^2 ds} W_j$$

```
// ExoticBSEngine.h
#ifndef EXOTIC_BS_ENGINE_H
#define EXOTIC_BS_ENGINE_H
#include <ExoticEngine.h>
#include <Random2.h>
class ExoticBSEngine : public ExoticEngine
{
public:
    ExoticBSEngine(const Wrapper<PathDependent>& TheProduct_,
        const Parameters& R_,
        const Parameters& D_,
        const Parameters& Vol_,
        const Wrapper<RandomBase>& TheGenerator_,
        double Spot_);
    virtual void GetOnePath(MJArray& SpotValues);
    virtual ~ExoticBSEngine(){}

private:
    Wrapper<RandomBase> TheGenerator;
    MJArray Drifts;
    MJArray StandardDeviations;
    double LogSpot;
    unsigned long NumberOfTimes;
    MJArray Variates;
};
#endif

// ExoticBSEngine.cpp
#include <ExoticBSEngine.h>
#include <cmath>
void ExoticBSEngine::GetOnePath(MJArray& SpotValues)
{
```

```
    TheGenerator->GetGaussians(Variates);
    double CurrentLogSpot = LogSpot;
    for (unsigned long j=0; j<NumberOfTimes; j++)
    {
        CurrentLogSpot += Drifts[j];
        CurrentLogSpot += StandardDeviations[j] * Variates[j];
        SpotValues[j] = exp(CurrentLogSpot);
    }
    return;
}
ExoticBSEngine::ExoticBSEngine(const Wrapper<PathDependent>& TheProduct_,
        const Parameters& R_,
        const Parameters& D_,
        const Parameters& Vol_,
        cosnt Wrapper<RandomBase>& TheGenerator_,
        double Spot_)
        :
        ExoticEngine(TheProduct_, R_),
        TheGenerator(TheGenerator_)
{
    MJArray Times(TheProduct_->GetLookAtTimes());
    NumberOfTimes = Times.size();
    TheGenerator->ResetDimensionality(NumberOfTimes);
    Drifts.resize(NumberOfTimes);
    StandardDeviations.resize(NumberOfTimes);
    double Variance = Vol_.IntegralSquare(0, Times[0]);
    Drifts[0] = R_.Integral(0.0, Times[0])
        -D_.Integral(0.0, Times[0]) - 0.5 * Variance;
    StandardDeviations[0] = sqrt(Variance);
    for (unsigned long j=1; j<NumberOfTimes; ++j)
    {
        double thisVariance = Vol_.IntegralSquare(Times[j-1], Times[j]);
        Drifts[j] = R_.Integral(Times[j-1], Times[j]) - D_.Integral(Times[j-1], Times[j]) -
        StandardDeviations[j] = sqrt(thisVariance);
    }
LogSpot = log(Spot_);
Variates.resize(NumberOfTimes);
}
```

For geometric Asian options it will be faster as it only incvolves 1 exponentiation instead of many.

7.6. Arithmetic Asian Option

```
// PathDependentAsian.h
#ifndef PATH_DEPENDENT_ASIAN_H
#define PATH_DEPENDENT_ASIAN_H
#include <PathDependent.h>
#include <PayOffBridge.h>
class PathDependentAsian : public PathDependent
{
public:
    PathDepenedntAsian(const MJArray& LookAtTimes_,
            double DeliveryTime_,
            const PayOffBridge& ThePayOff_);
    virtual unsigned long MaxNumberOfCashFlows() const;
    virtual MJArray PossibleCashFlowTimes() const;
    virtual unsigned long CashFlows(const MJArray& SpotValues,
            std::vector<CashFlow>& GeneratedFlows) const;
    virtual ~PathDependetAsian(){}
    virtual PathDependent* clone() const;
private:
    double DeliveryTime;
    PayOffBrige ThePayOff;
    unsigned long NumberOfTimes;
};
#endif

// PathependentAsian.cpp
#include <PathDependentAsian.cpp>
PathDependentAsian::PathDependentAsian(cosnt MJArray&
                LookAtTimes_,
            double DeliveryTime_,
            const PayOffBridge&ThePayOff_)
            :
            PathDependent(LookAtTimes_),
            DeliveryTime(DeliveryTime_),
            ThePayOff(ThePayOff_),
            NumberOfTimes(LookAtTimes_.size())
            {}
unsigned long PathDependentAsian::MaxNumberOfCashFlows() const
{
    return 1UL;
}
MJArray PathDependentAsian::PossibleCashFlowTimes() const
{
    MJArray tmp(1UL);
    tmp[0] = DeliveryTime;
```

```
    return tmp;
}
unsigned long PathDependentAsian::CashFlows(const MJArray& SpotValues,
     std::vector<CashFlow>& GeneratedFlows) const
{
    double sum = SpotValues.sum();
    double mean = sum/NumberOfTimes;
    GeneratedFlows[0].TimeIndex = OUL;
    GeneratedFlows[0].Amount = ThePayOff(mean);
    return 1UL;
}
PathDependent* PathDependentAsian::clone() const
{
    return new PathDependentAsian(*this);
}
```

7.7. Summary

- Design process needs to identify the necessary components and specify how they talk to each other
- Template pattern involves deferring implementation of important part of algorithm to inherited class
- If option class knows nothing that's not specified in term-sheet then it's much easier to reuse
- We can reuse `Payoff` class to simplify coding of our more complicated path-dependent derivatives

8. Trees

8.1. Intro

Value of S_t doesn't depend on path of W_t but solely on its value at t, only Y_l matters for each X_j, which means that our tree is recombining, it doesn't matter whether we go down then up or not, we assume vol to be const. See formula 8.6. We're approximating continuous martingales with discrete r.v. almost as martingales.

Tree is good for pricing American options, this corresponds to optimal strategy of exercise iff exercise gives more money than not.

American option pricing algorithm.

- Create array of final spot values $S_0 e^{(r-d-1/2\sigma^2)T+\sigma\sqrt{T/N}j}$, where $j \in [-N, N]$

- For each of spot values evaluate payoff and store it
- All previous time-slice compute possible values of spot: $S_0 e^{(r-d-1/2\sigma^2)(N-1)T/N+\sigma\sqrt{T/N}j}$, where $j \in [-N+1, N-1]$
- For each values of spot, compute payoff and take max with discounted payoff of 2 possible values of spot at next time.
- Repeat until time 0

For knock-out barrier option, procedure is the same as European except that value at a point in tree would be 0 if lay behind barrier.

8.2. `TreeProduct` class

```
// TreeProducts.h
#ifndef TREE_PRODUCTS_H
#define TREE_PRODUCTS_H
class TreeProduct
{
public:
    TreeProduct(double FinalTime_);
    virtual double FinalPayOff(double Spot) const = 0;
    virtual double PreFinalValue (double Spot,
                                  double Time,
                                  double DiscountedFutureValue)
                                  const = 0;
    virtual ~TreeProduct(){}
    virtual TreeProduct* clone() const = 0;
    double GetFinalTime() const;
private:
    double FinalTime;
};
#endif

// TreeProducts.cpp
#include <TreeProducts.h>
TreeProduct::TreeProduct(double FinalTime_)
    : FinalTime(FinalTime_) {}
double TreeProduct::GetFinalTime() const
{
    return FinalTime;
}
// TreeAmerican.h
#ifndef TREE_AMERICAN_H
#define TREE_AMERICAN_H
#include <TreeProducts.h>
```

```
#include <PayOffBridge.h>
class TreeAmerican : public TreeProduct
{
public:
    TreeAmerican(double FinalTime,
                 const PayOffBridge& ThePayOff_);
    virtual TreeProduct* clone() const;
    virtual double FinalPayOff(double Spot) const;
    virtual double PreFinalValue(double Spot,
                                 double Time,
                                 double DiscountedFutureValue) const;
    virtual ~TreeAmerican() {}
private:
    PayOffBridge ThePayOff;
};
#endif

// TreeAmerican.cpp
#include <TreeAmerican.h>
#include <minmax.h>
TreeAmerican::TreeAmerican(double FinalTime,
                        const PayOffBridge& ThePayOff_)
                        : TreeProduct(FinalTime),
                        ThePayOff(ThePayOff_)
                        {}
TreeProduct* TreeAmerican::clone() const
{
    return new TreeAmerican(*this);
}
double TreeAmerican::FinalPayOff(double Spot) const
{
    return ThePayOff(Spot);
}
double TreeAmerican::PreFinalValue(double Spot,double ,) const
{
    return max(ThePayOff(Spot), DiscountedFutureValue);
}
```

Because our products have time of expiry, so we implicitly disallowing perpetual options. We use `clone` and virtual destructor to allow virtual copying, and to ensure absence of memory leaks after virtual copying. The remaining methods are pure virtual and specify the value of product at expiry.

8.3. Tree class

We can price multiple products with the same expiry, we call method multiple times and only build tree once. So we store the entire tree.

```
// BinomialTree.h
#pragma warning (disable: 4786)
#include <TreeProduct.h>
#include <vector>
#include <Parameters.h>
#include <Arrays.h>
class SimpleBinomialTree
{
public:
    SimpleBinomialTree(double Spot_,
                       const Parameters& r_,
                       const Parameters& d_,
                       double Volatility_,
                       unsigned long Steps,
                       double Time);
    double GetThePrice(const TreeProduct& TheProduct);
protected:
    void BuildTree();
private:
    double Spot;
    Parameters r;
    Parameters d;
    double Volatility;
    unsigned long Steps;
    double Time;
    bool TreeBuilt;
    std::vector<std::vector<std::pair<double, double>>> TheTree;
    MJArray Discounts;
};
```

We store the tree as a vector of vectors of pairs of doubles, so we have `#pragma` at first, without which will get a warning message telling us the debug info is too long. `pair` is a simple template in STL which gives a class with 2 data members of appropriate types as first and second. An alternative implementation would be to have 2 trees, one for spot and another for option values. That will require twice as much work when resizing.

```
// BinomialTree.cpp
#include <BinomialTree.h>
#include <Arrays.h>
```

```
#include <cmath>
#if !defined(_MSC_VER)
using namespace std;
#endif
SimpleBinomialTree::SimpleBinomialTree(double Spot_,
                        const Parameters& r_,
                        const Parameters& d_,
                        double Volatility_,
                        unsigned long Steps_,
                        double Time_)
                        : Spot(Spot_),
                        r(r_),
                        d(d_),
                        Volatility(Volatility_),
                        Steps(Steps_),
                        Time(Time_),
                        Discounts(Steps)
{
    TreeBuilt = false;
}
void SimpleBinomialTree::BuildTree()
{
    TreeBuilt = true;
    TheTree.resize(Steps+1);
    double InitialLogSpot = log(Spot);
    for (unsigned long i = 0; i < Steps; i++)
    {
        TheTree[i].resize(i+1);
        double thisTime = (i * Time)/Steps;
        double movedLogSpot =
            InitialLogSpot + r.Integral(0.0, thisTime)
            - d.Integral(0.0, thisTime);
        movedLogSpot -= 0.5 * Volatility*Volatility*thisTime;
        double sd = Volatility * sqrt(Time/Steps);
        for (long j = -static_cast<long>(i), k=0;
        j<=static_cast<long>(i); j=j+2, k++)
        TheTree[i][k].first = exp(movedLogSpot + j * sd);
    }
    for (unsigned long l=0; l <Steps; l++)
    {
        Discounts[l] = exp(-r.Integtal(l*Time/Steps, (l+1)*Time/Steps));
    }
}
double SimpleBinomialTree::GetThePrice(const TreeProduct& TheProduct)
{
    if (!TreeBuilt)
```

```
        BuildTree();
    if (TheProduct.GetFinalTime() != Time)
        throw("mismatched product in simplebinomialtree");
    for (long j =- static_cast<long>(Steps)
        , k=0; j <= static_cast<long>(Steps); j=j+2, k++)
        TheTree[Steps][k].second =
            TheProduct.FinalPayOff(TheTree[Steps][k].first);
    for (unsigned long i=1; i<=Steps; i++)
    {
        unsigned long index = Steps-i;
        double ThisTime = index * Time/Steps;
        for (long j =- static_cast<long>(index), k=0;
        j <= static_cast<long>(index); j=j+2; j++)
        {
            double Spot = TheTree[index][k].first;
            double futureDiscountedValue = 0.5*Discounts[index]*
                (TheTree[index+1][k].second +
                TheTree[index+1][k+1].second);
                TheTree[index][k].second=
                TheProduct.PreFinalValue(Spot, ThisTime,
                    futureDiscountedValue);
        }
    }
    return TheTree[0][0].second;
}
```

8.4. Pricing on the tree

```
// PayOffForward.h

#ifndef PAY_OFF_FORWARD_H
#define PAY_OFF_FORWARD_H
#include <PayOff3.h>
class PayOffForward: public PayOff
{
public:
    PayOffForward(double Stike_);
    virtual double operator()(double Spot) const;
    virtual ~PayOffForward(){}
    virtual PayOff* clone() const;
private:
    double Strike;
};
#endif
```

```cpp
//PayOffForward.cpp
#include <PayOffForward.h>
double PayOffForward::operator() (double Spot) const
{
    return Spot-Strike;
}
PayOffForward::PayOffForward(double Strike_) : Strike(Strike_)
{}
PayOff* PayOffForward::clone() const
{
    return new PayOffForward(*this);
}
```

8.5. Summary

- Tree pricing is baed on discretization of Brownian motion, and is a natural way to price American options.
- On a tree, knowledge of discounted future values is natural but knowing about the past is not.
- We can reuse the payoff class when defining products on trees
- By having a separate class encapsulating the definition of a derivative on a tree, we can reuse the product for more general structures.
- Euro options can be used as controls for American options

9. Solvers, templates, implied vol

9.1. Function objects

We want to solve BlackScholes(vol) = price, by Newton Raphson. We can use engine template, we define a base class, but disadvantage is that:

- The function call is virtual, and have efficiency problems. Because to call a virtual function, the processor has to look up a virtual function table each time the function is called, and then jump to a location specified by the table. If we don't need to look at table then it'll be faster. Also, it's not possible to `inline` virtual functions. If the function is known, the compiler can inline it and eliminate the mechanics of the function call altogether.
- Inheriting from solver base class inhibits other inheritance. If we wish to inherit class defining our function from some other class, we can't inherit from solver class as well without using multiple inheritance, which is tricky to get to work in bug-free fashion, and we want to avoid it.

So we want to input a function to our routine without using virtual functions, we can use a function pointer but this would buy us little over virtual functions. Another way is templatization, and the type of function being used in optimization is decided at compile time rather than runtime. So the compiler cn carry out optimizations and inlining that depend on type of function since that info is now available to it.

```
const operator() (double x) const = f.operator()(y)
```

```
// BSCallClass.h
#ifndef BS_CALL_CLASS_H
#define BS_CALL_CLASS_H
class BSCall
{
public:
    BSCall(double r_, double d_,
            double T, double Spot_,
            double Strike_);
    double operator()(double Vol) const;
private:
    double r;
    double d;
    double T;
    double Spot;
    double Strike;
};
#endif
```

```
//BSCallClass.cpp
#include <BSCallClass.h>
#include <BlackScholesFormulas.h>
BSCall::BSCall(double r_, double d_,
                double T_, double Spot_,
                double Strike_)
                :
                r(r_), d(d_),
                T(T_), Spot(Spot_),
                Strike(Strike_)
        {}
double BSCall::operator()(double Vol) const
{
    return BlackScholesCall(Spot, Strike, r,d,Vol,T);
}
```

9.2. Newton-Raphson and funtion template arguments

```
// NewtonRaphson.h
template<class T, double (T::*Value)(double) const,
                  double (T::*Derivative)(double) const>
                  double NewtonRaphson(double Target,
                                       double Start,
                                       double Tolerance,
                                       const T& TheObject)
{
    double y = (TheObject.*Value)(Start);
    double x=Start;
    while (fabs(y-Target)>Tolerance)
    {
        double d = (TheObject.*Derivative)(x);
        x += (Target-y)/d;
        y = (TheObject.*Value)(x);
    }
    return x;
}
```

We have 3 template parameters: the class, pointer to value function for the class, and pointer to derivative function for that class.

9.3. Using Newton-Raphson to do implied vol

```
// BSCallTwo.h
#ifndef BS_CALL_TWO_H
#define BS_CALL_TWO_H
class BSCallTwo
{
public:
    BSCallTwo(double r_, double d_,
              double T, double Spot_,
              double Strike_);
    double Price(double Vol) const;
    double Vega(double Vol) const;
private:
    double r;
    double d;
    double T;
    double Spot;
    double Strike'
};
#endif
```

```
//BSCallTwo.cpp
#include<BSCallTwo.h>
#include <BlackScholesFormulas.h>
BSCallTwo::BSCallTwo(double r_, double d_,
                     double T_, double Spot_,
                     double Strike_)
                     :
                     r(r_), d(d_),
                     T(T_), Spot(Spot_),
                     Strike(Strike_)
                     {}
double BSCallTwo::Price(double Vol) const
{
    return BlackScholesCall(Spot, Strike, r,d,Vol, T);
}
double BSCallTwo::Vega(double Vol) const
{
    return BlackScholesCallVega(Spot, Strike, r,d,Vol,T);
}
```

9.4. Pros and Cons of templation

We use template to achieve resuability. Template argument types are decided at time of compilation, while virtual functions is not determined until runtime.

- Speed. No time is spent on deciding which code to run when the code is running. Compiler knows which code will be run, so it can make extra optimizations which is hard if not impossible when the decision is made at run time.
- Size. Code is compiled for each template argument used separately, we have many copies of similar code. For simple routine like a solver, it's not an issue. But for complicated routine, it will be a large executable. If we have several template parameters the size could multiply out of control. *Ex*: Monte Carlo path-dependent exotic pricer, we had templated both random number generator and product. If we had 6 random number generators and 10 products, and we want to allow any combination then we will have 60 times as much code.
- Harder for user of code to make choices. If user can choose number generator and product via outside input, we must write code that branched into each of 60 cases and within each branch called the engine and gathered results
- Harder to debug. Compilers will often not compile lines of template code that're not used, so if a templatized class has a method that's not called

anywhere, the code will compile even if it has syntax errors. Only when a line is addedthat calls the particular method will the compiler errors apprear. To avoid this, we can write non-template code for a particular choice of template parameter. This code can be thoroughly tested and debugged, and then the code can be rewritten by changing particular parameter into a template parameter.

When should we use templates and when use virtual functions?

We don't use templates unless certain conditions are met. For routines to be short and potentially reusable in totally unrelated contexts. *Ex*: use templates for a numerical integration routine and a solver. We can also use templates for a container class, and use templates given in the standard template library. But we won't use templates for an option pricing engine, since the code will be long and is only relevant to quite specific context.

Language exhibit a tradeoff between abstraction and efficiency, and C++ strives to achieve both. Templates are ultimately a way of achieving abstraction without sacrificing efficiency.

9.5. Summary

- Templates are an alternative to inheritance for coding without knowing an object's precise type.
- Template code can be faster as function calls are determined at compile time.
- Extensive use of template code can lead to very large executables.
- Pointers to member functions can be useful to obtain generic behavior.
- Implied vol can only be computed numerically.

10. Factory

10.1. Intro

We want to design an interface that user will input name of payoff and strike, and program will price vanilla option with that payoff.

We can write a function that takes in string and strike, check again all known types of payoff and when it comes across the right one, create a payoff of the right type, we implement this by a switch statement. But everytime we add a new payoff we need to modify the switch statement, so it's violating open-closed principle. We don't want to change any existing files, but only add new files to object. Factory pattern can manufacture objects.

We need each type of payoff to tell the factory that it exists, and to give factory a blueprint [3] for its manufacture.

How to get class to communicate with factory without explicitly calling anything from the main routine? Global variables, initialized when program commences before anything else happens. If we define a class to initialize a global variable of that class registers a payoff class with factory, then it's done. Initialization involves a call to a constructor, and we can make the constructor do whatever we want.

For each payoff class, we write auxiliary class whose constructor registers the payoff class with our factory, and we declare a global variable of auxiliary class. Since auxiliary class are similar to each other, we adopt a template solution to define these classes. We need a factory object for these auxiliary classes to talk to. The factory object can't be a global variable as we have no control over the order in which the global variables are initialized. We need it to exist before the other globals are defined as they refer to it. So, it's `static` variable, defined in a function persits from one call to the next, and only disappears when program exits. On the first call to registration function, the factory comes into existence. So all the registration function calls will register payoff blueprints with the same factory. And the creator function will need to have access to the same factory object as the registration function, and if factory is hidden inside the registration function this will be impossible. So we need *singleton pattern*.

10.2. Singleton Pattern

We define a factory via static variable since it must come into existence as soon as it's referred to when registering blueprints. We need a factory class to exist, and we don't want other factory objects to exist as they'll confuse matters, everything must be registered with and built by the same factory.

Singleton pattern creates a class that all constructors and assignment operates are made private, so that factories can only be created from inside methods of class, so we have firm control over existence of factory objects. We define a method that defines a class object as static variable, class `PayOffFactory`, class method `Instance`.

```
PayOffFactory& PayOffFactory::Instance()
{
    static PayOffFactory theFactory;
    return theFactory;
}
```

[3] An identity string to distinguish that class and a pointer to a function that will create object of that class.

The first time the `Instance` is called, it creates static data member `theFactory`. As a member function, it can use private default constructor. Every subsequent time the `Instance` is called, the address of alread-existing static variable `theFactory is returned`, so the `Instance` creates precisely 1 `PayOffFactory` object which can be accessed from anywhere by calling `PayOffFactory::Instance()`. static here means for a *function* is different from the meaning for *variable*: for a function it means that the function can be called direcly without any attachment to an object. Singleton means only 1 object from class exists.

10.3. Factory coding

```
// PayOffFactory.h
#ifndef PAYOFF_FACTORY_H
#define PAYOFF_FACTORY_H
#include <PayOff3.h>
#if defined (_MSC_VER)
#pragma warning (disable:4786)
#endif
#include <map>
#include <string>
class PayOffFactory
{
public:
    typedef PayOff* (*CreatePayOffFunction)(double );
    static PayOffFactory& Instance();
    void RegisterPayOff(std::string, CreatePayOffFunction);
    PayOff* CreatePayOff(std::string PayOffId, double Strike);
    ~PayOffFactory(){};
private:
    std::map<std::string, CreatePayOffFunction>
        TheCreatorFunctions;
    PayOffFactory(){}
    PayOffFactory(const PayOffFactory&) {}
    PayOffFactory& operator = (const PayOffFactory&){return *this;}
};
#endif

// PayOffFactory.cpp
#if defined(_MSC_VER)
#pragma warning(disable: 4786)
#endif
#include <PayOffFactory.h>
#include <iostream>
using namespace std;
```

```
void PayOffFactory::RegisterPayOff(string PayOffId,
            CreatePayOffFunction CreatorFunction)
{
    TheCreatorFunctions.insert(pair<string, CreatePayOffFunction>
            (PayOffId, CreatorFunction));
}
PayOff* PayOffFactory::CreatePayOff(string PayOffId, double Strike)
{
    map<string, CreatePayOffFunction>::const_iterator
        i  = TheCreatorFunctions.find(PayOffId);
    if (i == TheCreatorFunctions.end())
    {
        std::cout << PayOffId << "is an unknown payoff" << std::endl;
        return NULL;
    }
    return (i->second)(Strike);
}
PayOffFactory& PayOffFactory::Instance()
{
    static PayOffFactory theFactory;
    return theFactory;
}
```

STL map container

A `map` is a collection of `pairs`: pair class stores the nodes of a tree, consisting of 2 public data members `first` and `second`. Type of these data members are template parameters, `first` is the key or identifier used to look up the object we want to find which is stored in `second`. Type of the map is `map<std::string, CreatePayOffFunction>`, and `insert` method places pairs of strings and `CreatePayOffFunctions` into the map. A map's each key is unique, so if you insert 2 pairs with the same key, then the second one is ignored. So we can examine the return type of insert to determine whether the insertion was successful by `RegisterPayOff`. We use `CreatePayOff` to do retrieval, a string is passed in and we use `find` method of map. The method returns `const_iterator` pointing to the pair which has correct key (first element) if such a pair exists, and otherwise iterator [4] points to the end of map.

10.4. Automatic registration

```
// PayOffConstructible.h
#ifndef PAYOFF_CONSTRUCTIBLE_H
```

[4] Iterator is an abstraction of a pointer and works in a similar fashion. Just like pointers, they can be dereferenced via * or ->. A const_iterator is similar to non-const pointer to const objects. Iterator's value can be changed, but the value of the thing it points to can't.

```
#define PAYOFF_CONSTRUCTIBLE_H
#if defined (_MSC_VER)
#pragma warning(disable:4786)
#endif
#include <iostream>
#include <PayOff3.h>
#include <PayOffFavtory.h>
#include <string>
template <class T>
class PayOffHelper
{
public:
    PayOffHelper(std::string);
    static PayOff* Create(double);
};
template <class T>
PayOff* PayOffHelper<T>::Create(double Strike)
{
    return new T(Strike);
}
template <class T>
PayOffHelper<T>::PayOffHelper(std::string id)
{
    PayOffFactory& thePayOffFactory = PayOffFactory::Instance();
    thePayOffFactory.RegisterPayOff(id, PayOffHelper<T>::Create);
}
#endif

// PayOffRegistration.cpp
#include <PayOffConstructible.h>
namespace
{
    PayOffHelper<PayOffCall> RegisterCall('call')
    PayOffHelper<PayOffPut> RegisterPut('put');
}
```

10.5. Summary

- Factory and Singleton pattern to give method of adding new payoff classes to an interface without modifying existing files
- Singleton pattern allows us to create a unique global object from a class and provide a way to access it
- Factory pattern adds extra inherited classes to be accessed from an interface without changing any existing files
- Factory pattern can be implemented using singleton pattern

- Standard template library `map` class is a convenient way to associate objects with string identifiers
- Placing objects in an unnamed namespace is a way of ensuring that they aren'd accessed elsewhere
- We can achieve automatic registration of classes by making their registration a side-effect of the creation of global variables from a helper class

11. Design Pattern revisited

11.1. Creational patterns

We create new objects, and we want to abstract the creation process to help the system developed independently of types of individual objects.

11.1.1. Virtual copy constructor

We need a copy of an object, but don't know its type so we can't use the copy constructor so we ask object to provide the copy itself. Because object knows more about itself than we do, so we ask it to help us. We can modify this pattern to ask object to make a default object from its class, as once again it knows its class type and we don't, for virtual constructors it's called factory method.

11.1.2. Factory

Abstract factory pattern has an object that spits out objects as and when we need them. We have greater control over their creation, with greater flexibility in changing objects used. This pattern gives us an easily extended interface, and allows the addition of new classes to an interface without rewriting any code.

11.1.3. Singleton

We use singleton to implement factory. There is a single copy of the object, which is accessible from everywhere without introducing global variables. If we want more than one copy of the object to exist, then we can modify the pattern to give us a doubleton or tripleton.

11.1.4. Monostate

It's an alternative to singleton pattern. Rather than only allowing one object from the class to exist, we allow an unlimited number but make them all have

the same member variables. So all the objects from the class act as one. We can do so by making all data members static. So we can treat each object from the class like any other, althought they're all the same object.

11.2. Structural patterns

How classes are composed to define more intricate structures. This allows our code to have extra functionality without having to rewrite existing code.

11.2.1. Adapter

A class that translates an interface into a form that other classes expect.

We want to fit code into a structure for which it's not originally designed, because we have changed our way of doing things or the code originates elsewhere.

Ex: We download a library from web, its interface is unlikely to conform to what we are using. By adapting its interface, we can integrate it into existing code.

11.2.2. Bridge

Similar to adapter, it defines an interface, and acts as an intermediary between a client class and the classes implementing the interface. So the implementing class can be changed without the client class being aware of the change.

Difference between Adapter and Bridge: bridge is intended to define an intermediary interface from the start; adapter is introduced a later stage for solving incompatilibities.

11.2.3. Decorator

We can change the behavior of a class at runtime without changing its interface. A wrapper class process incoming or outgoing messages and then passes them on. We can decorate as many times as we like since the interface after decoration is the same as the interface before.

11.3. Behavioral patterns

To implement algorithms for reuse.

11.3.1. Strategy

We defer an important part of algorithm to an inputted object, which allows us to change how this particular part of algo behaves.

11.3.2. Template

Rather than inputting an aspect of the algorithm, we defer part of algorithm's implementation to an inherited class. The base class provides structure of how different parts of algorithm fits together, but doesn't specify all the details of implementation. *Ex*: we can price exotics using different model in the future.

11.3.3. Iterator

Iterator is an abstraction of a pointer, and we can dereference it, i.e. look at what it points to, increment and decrement it. Iterator can be defined for any sort of data structure. In STL, algorithms take type of interator as a template argument, allowing this generality to be implemented.

11.4. Summary

- Design patterns can be classified into behaviral, structural and creational patterns
- Behavioral patterns are used for implementing algorithms
- Structural patterns deal with how classes are composed to create more intricate designs
- Creation patterns deal with the creation of new objects

12. Exceptions

It's possible that we call `operator()` will throw an exception, which is caught somewhere outside function.

12.1. Two Safety guarantees

- weak guarantee: the object and program are left in a valid state, and no resources have been leaked
- strong guarantee: if an exception is thrown during an operation (a call to a method/function), then the program is left in the state it was at entry to the operation

For weak guarantee, an object's state can change even though operation failed, but with strong guarantee the class is promising to undo all changes before throwing.

12.2. Smart pointers

When function is exited, we want the memory allocated by clone command is deallocated by a call to delete. Exiting can occur either in conventional way via return, or by exception being thrown. For both, all automatic variables are destroyed at the end of scope.

Smart pointer `Wrapper<T>`

```
double evaluate(const PayOff& p,
                double y)
{
    Wrapper<PayOff> payOffPtr(p);
    double x = (*payOffPtr)(y);
    return x;
}
```

Wrapper class will call clone method internally. delete is no longer necessary because the destructor of Wrapper calls it automatically. So, Wrapper can't be used to take owernership of a raw pointer since it has no constructors that take pointers. But we can add to file an extra constructor in public section of the class.

```
Wrapper(T* DataPtr_)
{
    DataPtr = DataPtr_;
}
double evaluate(const PayOff& p,
                double y)
{
    PayOff* payPtr1 = p.clone();
    Wrapper<PayOff> payOffPtr(payPtr1);
    double x = (*payOffPtr)(y);
    return x;
}
```

4 solutions to copying:

1. Copy the pointed-to object
2. Make copying illegal
3. Have pointers share ownership of object
4. Transfer ownership of pointer to new object

Second: we make copy constructor and assignment operator of object private. We can use `scoped_ptr` class from Boost defined in `boost/scoped_prt.hpp`.

The error is generated at compile time rather than run time, since it arised from access permissions to class methods.

Third: `boost/scoped_prt.hpp`, we have a reference-counted pointer class. Every time `shared_ptr` is copied, a count of how many pointers there are to the object will ++1. Every time one is destroyed the count –1.

Fourth: `auto_ptr` in file memory.

```
double evaluate(const PayOff& p,
                double y)
{
    std::suto_ptr<PayOff> payPtr1 = p.clone();
    double z = (*payPtr1)(y);
    std::auto_ptr<PayOff> payPtr2(payPtr1);
    double x = (*payPtr1)(y);
    return x+z;
}
```

12.3. Rule of almost zero

Rule of three: if you define one of copy constrtuctor, assignment operator and destructor of a class, then you should define all three.

Rule of almost zero: you should always be in the case of not defining any of them.

How to avoid shallow copy problem? We can use smart pointer to ensure that a shallow copy is sufficient. Every data member will be either an ordinary object which can be copied, or a smart pointer which is copied and assigned in the fashion we have chosen. If we want objects to be shared beteween copies, we use `shared_ptr`; if we want to make copying illegal, we use `scoped_ptr`, if we want pointed-to objects to be cloned, we use `Wrapper`.

Smart pointer has no memory leak issues because it deletes the point-to objects when the compiler-generated destructor is called. We don't waste time writing copy constructors or assignment operators, and we don't need to remember to update them when changing the data members of the class. [5]

Rule of almost zero: There is one case that we must declare a destructor but only an empty one: every time we have a class with abstract methods, it's likely to be deleted via pointers to the base class and we must declare a virtual destructor.

12.4. Commands to never use

Never use `malloc`, `free`, `delete`, `new[]`, `delete[]`.

[5] forgetting to keep them inline with class data members is a common source of bugs.

If anything is created by `new` owned by a smart pointer, we should never use `delete`. If we want n objects of class Option, we just use `std::vector<Option> v(n);`, the memory will be deleted automatically when necessary. `vector` is a smart pointer owning an array of objects.

12.5. Making wrapper class exception safe

Assignment operator (bad code)

```
Wrapper& operator=(const Wrapper<T>& original)
{
    if (this != original)
    {
        if (DataPtr!=0)
            delete DataPtr;
        DataPtr = (original.DataPtr !=  0)
                    ? original.DataPtr->clone() :0;
    }
    return *this;
}
```

If call to `clone` method of `original` object passed in throws, we have a problem. We've already deleted `DataPtr` so the strong guarantee is violated. Moreover, any attempt to access the underlying object will be an attempt to access a dead object, and we will have a crash when `Wrapper` goes out of scope and a second attempt to delete `DataPtr`.

Good code

```
Wrapper& operator=(const Wrapper<T>& original)
{
    if (this != & original)
    {
        T* newPtr = (original.DataPtr != 0) ?
                    original.DataPtr->clone():0;
        if (DataPtr!=0)
            delete DataPtr;
        DataPtr = newPtr;
    }
    return *this;
}
```

12.6. Throwing in special functions

It's ok to throw error in constructor but never throw in destructor. Destructor is called only for fully constructed objects, so if an exception is thrown in constructor, the destructors for all data members are called but the destructor for object being created is not. If destructor carries out non-trivial operations like calling delete, we have a problem. We can follow the rule of almost zero and have a trivial destructor, which is safer in that exceptions could arise in unexpected places. We should never throw in a destructor.

12.7. Summary

- Exceptions can cause memory leaks
- Weak or basic exception safety guarantee: a program will be in a valid state after an exception is thrown
- Strong exception safety guarantee: if an exception is thrown during an operation, then the program will be left in the state it was in at the start of the operation
- Memory leaks can be avoided by smart pointers
- Rule of almost zero: never write code that requires non-trivial copy constructors, assignment operators, and destructors
- Avoide `new[], delete, delete[]` commands
- Take care when writing assignment operators of smart pointers to avoid memory leaks when `new` fails
- Floating point errors don't by default cause C++ exceptions but they can be made to do so.

13. Templatizing the factory

13.1. Use inheritance to add structure

Key factor of our factory is singleton, key part of the singleton is that it can't be copied, we can do this by making all constructors including the copy constructor private. Inherited class's public interface doens't contain the public part of the base class.

13.2. Curiously recurring template pattern

We want to implement a reusable singleton via inheritance. How to implement the method that returns the sole instance of the class? We can use curiously recurring template pattern to templatize on the type of the inherited class.

```
template<class T>
class Singleton : private noncopyable
{
public:
    static T& Instance()
    {
        static T one;
        return one;
    }
protected:
    Singleton() {}
};
class MyFactory: public Singleton<MyFactory>
{
private:
    MyFactory() {}
    friend class Singleton<MyFactory>;
};
```

The friend declaration, the constructor for `MyFactory` is private, so we need to allow `Singleton` class to create the one object from `MyFactory`.

13.3. Using argument lists

```
// ArgList.h
#ifndef ARG_LIST_H
#define ARG_LIST_H
#include <xlw/port.h>
#include "CellMatrix.h"
#include "MyContainers.h"
#include <map>
#include <string>
#include <vector>
void MakeLowerCase(std::string& input);
class ArgumentList
{
public:
    ArgumentList(CellMatrix cells,
                 std::string ErrorIdentifier);
    ArgumentList(std::string name);
    enum ArgumentType
    {
        string, number, vector, matrix, boolean, list, cells
    };
    std::string GetStructureName() const;
```

```
    const std::vector<std::pair<std::string, ArgumentType>>&GetArgumentNamesAndTypes() const;
    std::string GetStringArgumentValue(
        const std::string& ArgumentName
    );
    double GetDoubleArgumentValue(const std::string& ArgumentName);
    MyArray GetArrayArgumentValue(const std::string& ArgumentName);
    // .... bool, void,
private:
    std::string StructureName;
    std::map<std::string, MyArray> ArrayArguments;
    // ...
};
#endif
```

This class allows data from 7 types with an arbitrarily large amount of data. Data is retrieved by string as a key.

13.4. Using templatized factory

```
// PayOff.h
#ifndef PAYOFF_H
#define PAYOFF_H
class PayOff
{
public:
    PayOff();
    virtual double operator()(double Spot) const = 0;
    virtual ~PayOff();
    virtual PayOff* clone() const = 0;
private:
};
#endif

//PayOffConcrete.h
#ifndef PAYOFF_CONCRETE_H
#define PAYOFF_CONCRETE_H
#include "PayOff.h"
#include <xlw/ArgList.h>
#include <xlw/Wrapper.h>
#include <xlw/ArgListFactory.h>
class PayOffCall: public PayOff
{
public:
    PayOffCall(ArgumentList args);
    virtual double operator()(double Spot) const;
```

```
    virtual ~PayOffCall(){}
    virtual PayOff* clone() const;
private:
    double Strike;
};
class PayOffPut:public PayOff
{
public:
    PayOffPut(ArgumentList args);
    virtual double operator()(double Spot) const;
    virtual ~PayOffPut(){}
    virtual PayOff* clone() const;
private:
    double Strike;
};
class PayOffSpread: public PayOff
{
public:
    PayOffSpread(ArgumentList args);
    virtual double operator()(double Spot) const;
    virtual ~PayOffSpread(){}
    virtual PayOff* clone() const;
private:
    Wrapper<PayOff> OptionOne;
    Wrapper<PayOff> OptionTwo;
    double Volume1;
    double Volume2;
};
#endif

// PayOffConcrete.cpp
// class PayOffSpread is an example of composite pattern,
// which is similar to decorator, difference is that more than 1 underlying class
#include <xlw/port.h>
#include "PayOffConcrete.h"
PayOffCall::PayOffCall(ArgumentList args)
{
    if (args.GetStructureName() != "payoff")
    // must be lower case here throw exception
    if (args.GetStringArgumentValue("name") != "call")
        throw("payoff list not for call passed to PayOffCall" :
        " got "+args.GetStringArgumentValue("name"));
    Strike = args.GetDoubleArgumentValue("strike");
    args.CheckAllUsed("PayOffCall");
}
double PayOffCall::operator() (double Spot) const
```

```
{
    return Spot-Strike > 0.0 ? Spot-Strike : 0.0;
}
PayOff* PayOffCall::clone() const
{
    return new PayOffCall(*this);
}
double PayOffPut::operator() (double Spot) const
{
    return Strike-Spot > 0.0 ? Strike-Spot :0.0;
}

// ......
```

To get underlying options, we use list arguments from `ArgumentList` passed in and call the same factory. The factory returns raw pointers, but these are immediately taken over by `Wrapper` class, which ensures that they're properly memory managed.

Important synergis between composite pattern and `ArgumentList` class: we can bring composite into factory because it's legitimate to have data stored in `ArgumentList` class, which is of the same type, and can used to create more objects from factory.

13.5. Summary

- Private inheritance can be used to express "implemented in terms of".
- Recurring template pattern can be used to make the return type of a base class method equal to type of inherited class.
- Singleton can be implemented using curiously recurring template pattern.
- We can use an argument list class to encapsulate a variable number of arguments of varying types.
- Some compilers have problems with curiously recurring template pattern implementaion of the singleton.
- The argument list class allows us to create a templatized factory without worrying about the types of arguments.
- The `CellMatrix` class gives us a way of transferrring data to and from spread sheets without having to deal with particulars of the spreadsheet's data types.

14. Interfacing with EXCEL

- `xlw` gives an easy way to create xlls

- xlls are a standard way of interfacing with EXCEL
- Quants generally don't use console applications
- All interfacing code is generated automatically by xlw.

15. Decoupling

A library is too long to rebuild. We need physical design, and insulation, a much stronger requirement than encapsulation: if A insulated from B, then changes to A don't cause B to recompile. But encapsulation only guarantees that B won't have to be recorded.

15.1. `inline`

`inline` is used for optimization, it gives compiler the option of replacing a function call with actual code defining the function. This eliminates a function call, thus saving time.

Usage: `inline` is for data access operations. *Ex*: when defining an array class, it's common to make `operator[]` an inline method, thus encapsulation of private data is preserved with no speed cost at run-time.

While we preserved encapsulation at no runtime cost, we have lost a lot of insulation at plenty of compile- and link-time cost. For inline to work, the function must be provided in the header file; thus we have dependency on implementation of the function or method. If we change function's implementation, all clients must recompile. With a non-inline function defined in source file, the impact is small.

If a function is non-trivial and heavily used, there will be many copies of the function sprinkled throughout the code, which will increase executable size. So, inline is useful for avoiding performance overheads with private data, we should also not overuse it.

15.2. Summary

- Physical design relates to how files depend on each other
- Encapsulation stops us from having to rewrite code
- Insulation saves us from having to recompile code
- Being careful with `#include` can speed up compile times
- Levelization yields a natural way to eliminate excess dependencies
- Inlining has costs in terms of physical design as well as gains in terms of run speed
- Template code increases coupling

- PIMPL is a powerful methodology for minimizing dependencies

---

𝔼nd - 𝕆f - 𝕋he - 𝔹ook - ℕotes