Design Patterns & Algorithms

Goals:

- Review polymorphism and virtual functions
- Discuss some of the most common object-oriented programming design patterns
- Review a few of the most common sort & search algorithms

Review: Polymorphism

- Recall from our last lecture that the use of **virtual functions** and **polymorphism** in a class hierarchy enables us to write completely generic, scalable code.
- In particular, we saw that if we defined our functions as virtual, then we can declare a variable in the base class, initialize that variable to one of many derived classes, and have the function call dynamically chosen at run-time to match the class that was initialized.
- This is an important concept, so let's look at another example before we move on...

Review: Polymorphism Example

```
1  class StochasticProcess
2  {
3  public:
4      double S0;
5      StochasticProcess(double S0_) { S0 = S0_; };
6          virtual double simulate() {
7          std::cout << "simulating from Base"
8                      << std::endl;
9          return 0.0;
10     };
11 };
12
13 class BlackScholesStochProcess : public StochasticProcess
14 {
15 public:
16     BlackScholesStochProcess(double S0_)
```

```
17          :StochasticProcess(S0_){};
19      virtual double simulate(){
20          std::cout << "simulating from Black Scholes"
21                    << std::endl;
22          return 0.0;
23      }
24  };
25
26  class BachelierStochProcess : public StochasticProcess
27  {
28      public:
29          BachelierStochProcess(double S0_)
30              :StochasticProcess(S0_){};
31          virtual double simulate(){
32              std::cout << "simulating from Bachelier"
33                        << std::endl;
34              return 0.0;
35          }
36  };
37
```

```
38 class CallOptionPricer
40 {
41 public:
42     std::auto_ptr<StochasticProcess> stochProc;
43     double shares;
44     double K;
45
46     CallOptionPricer(std::auto_ptr<StochasticProcess> sp_){
47         stochProc = sp_;
48     }
49
50     virtual double price(){
51         double valAtExp = stochProc->simulate();
52         return std::max(valAtExp-K, 0.0);
53     }
54 };
55
56 int main(int argc, const char * argv[]) {
57     int stochProcType = atoi(argv[1]);
58     double S0 = atof(argv[2]);
```

```
59
61      std::auto_ptr<StochasticProcess> stochProc;
62      if (stochProcType == 0){
63          stochProc = std::auto_ptr<StochasticProcess>
64              (new StochasticProcess(S0));
65      } else if (stochProcType == 1){
66          stochProc = std::auto_ptr<StochasticProcess>
67              (new BlackScholesStochProcess(S0));
68      } else if (stochProcType == 2){
69          stochProc = std::auto_ptr<StochasticProcess>
70              (new BachelierStochProcess(S0));
71      } else{
72          std::cout << "unexpected type" << std::endl;
73      return 1;
74      }
75      CallOptionPricer optPxer(stochProc);
76      optPxer.price();
77      return 0;
78  }
```

Comments on Polymorphism

- The key point here is that I have created a piece of code that can price instruments via simulation and I'm able to specify the stochastic process at run-time.
- This means that I can switch between stochastic processes without changing code but only by changing configuration parameters.
- An important side effect of this is that the code doesn't need to be re-built when switching stochastic processes.
- In order to add functionality for a new stochastic process, we just need to define another class that inherits from the StochasticProcess base class.

Design Patterns

- C++, as well as many other programming languages (i.e. Java) have a corresponding set of design patterns that help us solve certain problems in a generic way.
- We will only discuss a few of the many existing design patterns to give you a sense of what they are meant to accomplish.
- For those interested in exploring further, more information on design patterns in C++ can be found here: C++ Design Patterns

Design Patterns

- Design Patterns are broken into 3 logical groups:
  - **Creational**: deals with optimal creation/initialization of objects.
  - **Structural**: deals with organizing classes to form larger objects that provide new functionality.
  - **Behavioral**: deals with communications between two or more objects.

Singleton Pattern

- The **Singleton Pattern** is a *creational pattern* that ensures that **only one instance of a class** can be created each time the application is run.
- This paradigm is useful for things that we want to make sure are unique, such as conenctions to databases and log files.
- The Singleton Pattern is implemented by making a class's constructor **private** and defining a **static getInstance method** that checks whether the class is initialized and creates it only if it has not been initialized.
- The member variable should also be declared as static in the singleton pattern.

Singleton Pattern: Example

```
1  class DBConnection
2  {
3      public:
4      static std::shared_ptr<DBConnection> getInstance()
5      {
6          if (!db_conn)
7              db_conn = std::shared_ptr<DBConnection>
8                  (new DBConnection());
9              return db_conn;
10         }
11
12     private:
13         static std::shared_ptr<DBConnection> db_conn;
14         DBConnection() {}
15 };
```

Factory Pattern

- The **Factory Pattern** is useful for helping to create a class from a class hierarchy with many derived classes.
- The Factory Pattern is often paired with virtual functions & polymorphism to dynamically create different types of derived classes based on a program's settings or parameterization.
- The Factory Pattern enables us to decide at run-time the type of object we would like to create.
- The Factory Pattern consists of a **single static function** that returns the base class type and initializes the appropriate derived type based on a given input parameter.
- I often use an enum in conjunction with my Factory classes for more readable code.

Factory Pattern: Example

Let's look at an example of a Factory class using our generic Stochastic Process class example:

```
enum StochProc { Base_StochProc = 1,
                 BS_StochProc = 2,
                 Bachelier_StocProc = 3};

class StochProcFactory{
    public:
    static std::shared_ptr<StochasticProcess>
        createStochProc(StochProc sp, double S0){

            std::shared_ptr<StochasticProcess> stochProc;

            if (sp == Base_StochProc){
                stochProc = std::auto_ptr<StochasticProcess>
```

```
14                          (new StochasticProcess(S0));
16                  } else if (sp == BS_StochProc){
17                      stochProc = std::auto_ptr<StochasticProcess>
18                          (new BlackScholesStochProcess(S0));
19                  } else if (sp == Bachelier_StocProc){
20                      stochProc = std::auto_ptr<StochasticProcess>
21                          (new BachelierStochProcess(S0));
22                  }
23
24                  return stochProc;
25              }
26  };
```

Observer Pattern

- The observer pattern **enables communication between two classes**, defined as an **Observer** and an **Observable**.
- In particular, we use the observer pattern when we want to notify one or more observers of changes that we make to the observable.
- This paradigm is a useful way for making sure that all instances of an object are synchronized.
- As an example, we may use a yield curve to price a portfolio, and want to make sure that the same yields are used in all calculations.
- In this case we can use the observer pattern to make sure that all instances of a yield curve are updated at the same time.
- We could then ensure that the portfolio's positions are recalculated when an update to the yield curve is received.

QuantLib Observer Pattern

The QuantLib financial library has an implementation of the **Observer Pattern**, which is structured as follows:

```
class Observable {
    friend class Observer;
    public:
        void notifyObservers() {
            for (iterator i=observers_.begin();
            i!=observers_.end(); ++i) {
                try {
                    (*i)->update();
                } catch (std::exception& e) {
                    // store information for later
                }
            }
        }
```

```
14      private:
16          void registerObserver(Observer* o) {
17              observers_.insert(o);
18          }
19          void unregisterObserver(Observer*);
20              list<Observer*> observers_;
21          };
22
23  class Observer {
24      public:
25          virtual ~Observer() {
26              for (iterator i=observables_.begin();
27              i!=observables_.end(); ++i)
28                  (*i)->unregisterObserver(this);
29              }
30          void registerWith(const shared_ptr<Observable>& o) {
31              o->registerObserver(this);
32              observables_.insert(o);
33          }
34          void unregisterWith(const shared_ptr<Observable>&);
```

```
35          virtual void update() = 0;
37      private:
38          list<shared_ptr<Observable> > observables_;
39  };
```

- If we were using this pattern in QuantLib, we could define classes that inherited from the Observer / Observable classes respectively.
- While this is a powerful design pattern, it comes with great complexity and if not used correctly can make code significantly less efficient.
- Nonetheless it is a code concept to be aware of...

Template Method

- The **template method** works when we **define a sketch of an algorithm in an abstract base class** and then **declare certain key functions as pure virtual**.
- All derived classes will then be required to implement these pure virtual functions and will then be able to run the larger algorithm using the sketch from the base class.
- The benefit of this method is that we only need to code the sketch of the larger algorithm once in the base class.
- As an example, we could use the template method to define a base simulation class, with a pure virtual function called nextStep which we could then define in derived simulation classes (such as Black-Scholes Simulation)

Template Method: Example

```
class Simulate{
public:
    Simulate(double r_, double q_, double tau_){
        r = r_; q = q_; tau = tau_;
    }
    std::vector<double> runSimulation(double S0, int draws){
        int timesteps = tau * 252;
        std::vector<double> vec_ST;
        for (unsigned int ii = 0; ii < draws; ii++){
            double Shat = S0;
            for (unsigned int jj = 0; jj < timesteps; jj++){
                double next_dwt = getNextStep();
                Shat += next_dwt;
            }
            vec_ST.push_back(Shat);
        }
```

```
17          return vec_ST;
19      }
20      virtual double getNextStep() = 0;
21      double r, q, tau;
22  };
23
24  class BS_Simulate : public Simulate {
25  public:
26      BS_Simulate(double r_, double q_, double tau_, double sigma_)
27      :Simulate(r_, q_, tau_){
28          sigma = sigma_;
29      }
30      virtual double getNextStep(){
31          std::cout << "generate next BS step"
32                    << std::endl;
33          return 0.0;
34      }
35      double sigma;
36  };
```

Search Algorithms

- There are many ways to search for a value in an array.
- The simplest such way would just to iterate through every element in the array and compare the value of the element to the value we are searching for.
- This is a naive search algorithm as it requires us to perform the comparison on all array elements.
- An alternative method, on an already sorted array, is **binary search**.

Binary Search Algorithm

- The binary search algorithm is a way of searching for a value in an *already sorted* array.
- Binary search works in a similar way to bisection in root finding.
- Essentially we split the sorted array in half at each step, and check whether the item in the middle is below or above the value we are searching for.
- We can write a binary search algorithm in a generic way using a **templated function**.
- If we do this, the binary search function will work on any class as long as the comparison operators that we use in the definition are defined.

Binary Search Algorithm: Example

```
1  template <typename COMP_TYPE>
2  int BinarySearch( const std::vector<COMP_TYPE> & a,
3                    const COMP_TYPE & x )
4  {
5      int first = 0;
6      int last = a.size( ) - 1;
7
8      while( first <= last )
9      {
10         int mid = ( first + last ) / 2;
11
12         if( a[ mid ] < x ){
13             first = mid + 1;
14         }
15         else if( a[ mid ]>x){
16             last = mid - 1;
```

```
17            }
19            else{
20                return mid;
21            }
22        }
23        return -1;
24    }
```

NOTE: You should notice that we could easily replace the while loop in the Binary Search Algorithm with a recursive function call (How?). This algorithm is called Recursive Binary Search.

C++ Sort Algorithms

- Sorting an array in C++, or any other programming language, can be done in numerous ways, using many different **sorting algorithms**.
- Some of the more naive algorithms end up being less efficient, because they require traversing through the array multiple times.
- Sorting algorithms for arrays, or other structures are frequently used in technical quant interviews, so it's worth getting to know them a bit.
- Implementing sorting algorithms via templates gives us the ability to write generic sorting algorithms, as we saw in our binary search function.

C++ Sort Algorithms

- A few of the most common sorting algorithms are:
  - Selection sort
  - Insertion sort
  - Binary Search Insertion Sort
  - Merge Sort
  - Bubble Sort
  - Quick Sort

Selection Sort

- The **Selection Sort algorithm** is a naive sort algorithm that sorts by **repeatedly finding the smallest value in the array** and placing it at the beginning of the array.
- To do this, if there are n elements in the array, then **we need to loop through the array n times** in order to perform this sort algorithm.
- A selection sort algorithm is very easy to implement via a **nested for loop**.

Insertion Sort

- An insertion sort works by iterating sequentially through the cards and placing the chosen card in the appropriate place in the already sorted portion of the array.
- This algorithm requires that we perform one loop through the entire array and for each iteration of the loop we must also find it's appropriate place in the smaller sorted array.
- The most naive implementation of this would be to simply iterate through the smaller array as well until we find the right location of the chosen element.
- This approach would require a nested for loop, or a while loop within a for loop.

Insertion Sort: Example

```
1  void InsertionSort(int arr[], int n)
2  {
3      for (unsigned int i = 1; i < n; i++)
4      {
5          int val = arr[i];
6          int j = i-1;
7
8          while (j >= 0 && arr[j] > val)
9          {
10             arr[j+1] = arr[j];
11             j = j-1;
12         }
13         arr[j+1] = val;
14     }
15 }
```

Binary Search Insertion Sort

- An insertion sort algorithm can be improved by finding the right location of each element in the array more efficiently.
- One way to do this is to use **Binary Search** to find the appropriate location for each element.
- This could be implemented by replacing the while loop in the above slide with a call to the templated Binary Search function that we created on slide (24)
- One might also notice that we could also replace the outer for loop in our insertion sort algorithm with a recursive function call. (How?)

Bubble Sort

- A Bubble Sort is a simple sorting algorithm that works by repeatedly swapping elements that are sorted incorrectly until all elements are sorted in the correct order.
- Because this algorithm only moves elements one place at a time, it is not an efficient algorithm and many comparisons need to be done in order for the algorithm to know it is finished.
- In particular, the algorithm only knows that the array is sorted if a loop finishes without swapping any elements, which can be the exit criteria for our bubble sort algorithm.

Bubble Sort: Example

```
1  void bubbleSort(int arr[], int n)
2  {
3      bool has_swap = false;
4      for (unsigned int i = 0; i < n-1; i++){
5          for (unsigned int j = 0; j < n-i-1; j++) {
6              if (arr[j] > arr[j+1]) {
7                  std::swap(arr[j], arr[j+1]);
8                  has_swap = true;
9              }
10         }
11
12         if (!has_swap){
13             break;
14         }
15     }
16 }
```

Merge Sort

- Merge sort uses a divide and conquer algorithm and works by dividing the array in half, and calling merging the two halves after they have been sorted.
- A merge sort algorithm consists of two functions, a merge function that defines how to merge two sorted halves (and preserve the sort order) and an outer mergesort function that is called recursively.

Merge Sort: Example

```
1  void merge(int arr[], int l, int m, int r)
2  {
3      int n1 = m - l + 1;
4      int n2 =  r - m;
5      int L[n1], R[n2];
6      for (unsigned int i = 0; i < n1; i++){
7          L[i] = arr[l + i];
8      }
9      for (unsigned int j = 0; j < n2; j++){
10         R[j] = arr[m + 1+ j];
11     }
12     int i = 0;
13     int j = 0;
14     int k = l;
15     while (i < n1 && j < n2)
16     {
```

```
17          if (L[i] <= R[j])
19          {
20              arr[k] = L[i];
21              i++;
22          }
23          else
24          {
25              arr[k] = R[j];
26              j++;
27          }
28          k++;
29      }
30      while (i < n1)
31      {
32          arr[k] = L[i];
33          i++;
34          k++;
35      }
36      while (j < n2)
37      {
```

```
38          arr[k] = R[j];
40          j++;
41          k++;
42      }
43  }
44
45  void mergeSort(int arr[], int l, int r)
46  {
47      if (l < r)
48      {
49          int m = (l+r)/2;
50          mergeSort(arr, l, m);
51          mergeSort(arr, m+1, r);
52          merge(arr, l, m, r);
53      }
54  }
```