NYU | TANDON SCHOOL OF ENGINEERING

Topic 5:
Data Structures - Pricing American Options

Overview

- Introduce advanced object-oriented programming techniques and demonstrate how these techniques could be applied to solve complicated problems in quantitative finance.
- Implement American options by using C++ data structures, inheritance and templates.

American Options

- The holder of an ***American option*** has the right to exercise it at any time up to and including the expiry date ***N***. If the option is exercised at time step ***n*** and node ***i*** of the binomial tree, then the holder will receive payoff ***h(S(n, i))***.
- The price H(n, i) of an American option at any time step ***n*** and node ***i*** in the binomial tree can be computed by the following procedure, which proceeds by backward induction on ***n***:

- At the expiry date *N*: ***H(N, i) = h(S(N, i))***, for each node i = 0, 1, …, N.
- If H(n+1, i) is already known at each node i = 0, 1, …, n+1 for some n = 0, …, N-1, then

$$H(n,i) = \max\left(\frac{qH(n+1,i+1) + (1-q)H(n+1,i)}{1+R}, h(S(n,i))\right)$$

  for each node i = 0, 1, …, n.
- In particular, H(0) at the root node of the tree is the price of the American option at time 0.
- The Snell envelope procedure, from Discrete Models of Financial Markets, Marek Capinski and Ekkehard Kopp

Options06.h

```
#pragma once
#include "BinomialTreeModel02.h"
namespace fre {
	class Option
	{
	private:
		Option() : N(0) {}
		Option(const Option& option) : N(option.N) {}
	protected:
		int N;
	public:
		Option(int N_) : N(N_) {}
		int GetN() const { return N; }
		virtual double Payoff(double z) const = 0;
		virtual ~Option() = 0;
	};
```

Options06.h (continue)

```
class Call : public Option
{
private:
        double K;
public:
        Call(int N_, double K_) : Option(N_), K(K_) {}
        ~Call() {}
        double Payoff(double z) const;
};
class Put : public Option
{
private:
        double K;
public:
        Put(int N_, double K_) : Option(N_), K(K_) {}
        ~Put() {}
        double Payoff(double z) const;
};
```

Options06.h (continue)

```
class OptionCalculation
{
private:
    Option* pOption;
    OptionCalculation() : pOption(0) {}
    OptionCalculation(const OptionCalculation& optionCalculation) :
                                        pOption(optionCalculation.pOption) {}
public:
    OptionCalculation(Option* pOption_) : pOption(pOption_) {}
    ~OptionCalculation() {}
    double PriceByCRR(const BinomialTreeModel & Model);
    double PriceBySnell(const BinomialTreeModel & Model);
};
}
```

Options06.cpp

```
#pragma once
#include <iostream>
#include <cmath>
#include <vector>
#include "Option06.h"
#include "BinomialTreeModel02.h"
using namespace std;
namespace fre {
   Option::~Option() {}
   double Call::Payoff(double z) const
   {
      if (z > K) return z - K;
      return 0.0;
   }
   double Put::Payoff(double z) const
   {
      if (z < K) return K - z;
      return 0.0;
   }
```

```
double OptionCalculation::PriceByCRR(const BinomialTreeModel & Model)
{
   double q = Model.RiskNeutProb();
   int N = pOption->GetN();
   vector<double> Price(N + 1);
   for (int i = 0; i <= N; i++)
   {
      Price[i] = pOption->Payoff(Model.CalculateAssetPrice(N, i));
   }
   for (int n = N - 1; n >= 0; n--)
   {
      for (int i = 0; i <= n; i++)
      {
         Price[i] = (q * Price[i + 1] + (1 - q) * Price[i]) / Model.GetR();
      }
   }
   return Price[0];
}
```

```
double OptionCalculation::PriceBySnell(const BinomialTreeModel & Model)
{
   double q = Model.RiskNeutProb();
   int N = pOption->GetN();
   vector<double> Price(N + 1);
   double ContVal = 0;
   for (int i = 0; i <= N; i++)
   {
      Price[i] = pOption->Payoff(Model.CalculateAssetPrice(N, i));
   }
   for (int n = N - 1; n >= 0; n--)
   {
      for (int i = 0; i <= n; i++)
      {
         ContVal = (q * Price[i + 1] + (1 - q) * Price[i]) / Model.GetR();
         Price[i] = pOption->Payoff(Model.CalculateAssetPrice(n, i));
         if (ContVal > Price[i]) Price[i] = ContVal;
      }
   }
   return Price[0];
}
}
```

OptionPrice06.cpp

```
#include "BinomialTreeModel02.h"
#include "Option06.h"
#include <iostream>
#include <iomanip>
using namespace std;
using namespace fre;
int main()
{
	int N = 8;
	double U = 1.15125, D = 0.86862, R = 1.00545;
	double S0 = 106.00, K = 100.00;
	BinomialTreeModel Model(S0, U, D, R);

	Call call(N, K);
	OptionCalculation callCalculation(&call);
	cout << "European call option price = "
		<< fixed << setprecision(2) << callCalculation.PriceByCRR(Model) << endl;
```

```
    Put put(N, K);
    OptionCalculation putCalculation(&put);
    cout << "European put option price = "
         << fixed << setprecision(2) << putCalculation.PriceByCRR(Model) << endl;

    cout << "American call option price = "
         << fixed << setprecision(2) << callCalculation.PriceBySnell(Model) << endl;
    cout << "American put option price = "
         << fixed << setprecision(2) << putCalculation.PriceBySnell(Model) << endl;

    return 0;
}

/*
European call option price = 21.68
European put option price = 11.43
American call option price = 21.68
American put option price = 11.72
*/
```

- Notes:
  - A new member function PriceBySnell() is introduced in the class OptionCalculation for calculating the American call and put options, in other words, based on which pricing function is invoked, the call and put options could be considered as either European or American.
  - The Snell envelope procedure is implemented in the PriceBySnell to return the option price at time 0. The difference between PriceBySnell() and PriceByCRR() is the early exercise for American options.
  - In OptionPricer06, dynamic allocation for call and put is replaced by static allocated call and put object for simplicity.

- Notes:
  - An interesting feature used in the PriceBySnell() function (as well as PriceByCRR()) is the ***vector<> template*** from the ***Standard Template Library (STL):***
    - #include <vector>, loads the appropriate library to make vectors available in the program.
    - In addition to vectors, the STL contains many other useful predefined data structures we will talk in the details later.
  - **vector<double> Price(N+1);**
    - Declares a vector object comprising N+1 variables of type double.
  - **Price[i] = pOption->Payoff(Model.CalculateAssetPrice(N, i));**
    - Component i of the vector is referred to by Price[i] in the code, similar to using an array of type double.

Standard Template Library

- In the late 70s Alexander Stepanov first observed that some algorithms do not depend on some particular implementation of a data structure but only on a few fundamental semantic properties of the structure
- Developed by Stepanov and Lee at HP labs in 1992
- Become part of the C++ Standard in 1994

What's in STL

- Container classes: vector, list, stack, set, map, and etc…
- A large collection of algorithms, such as reverse, sort, search, and etc.

STL – C++ *Standard Template Library*

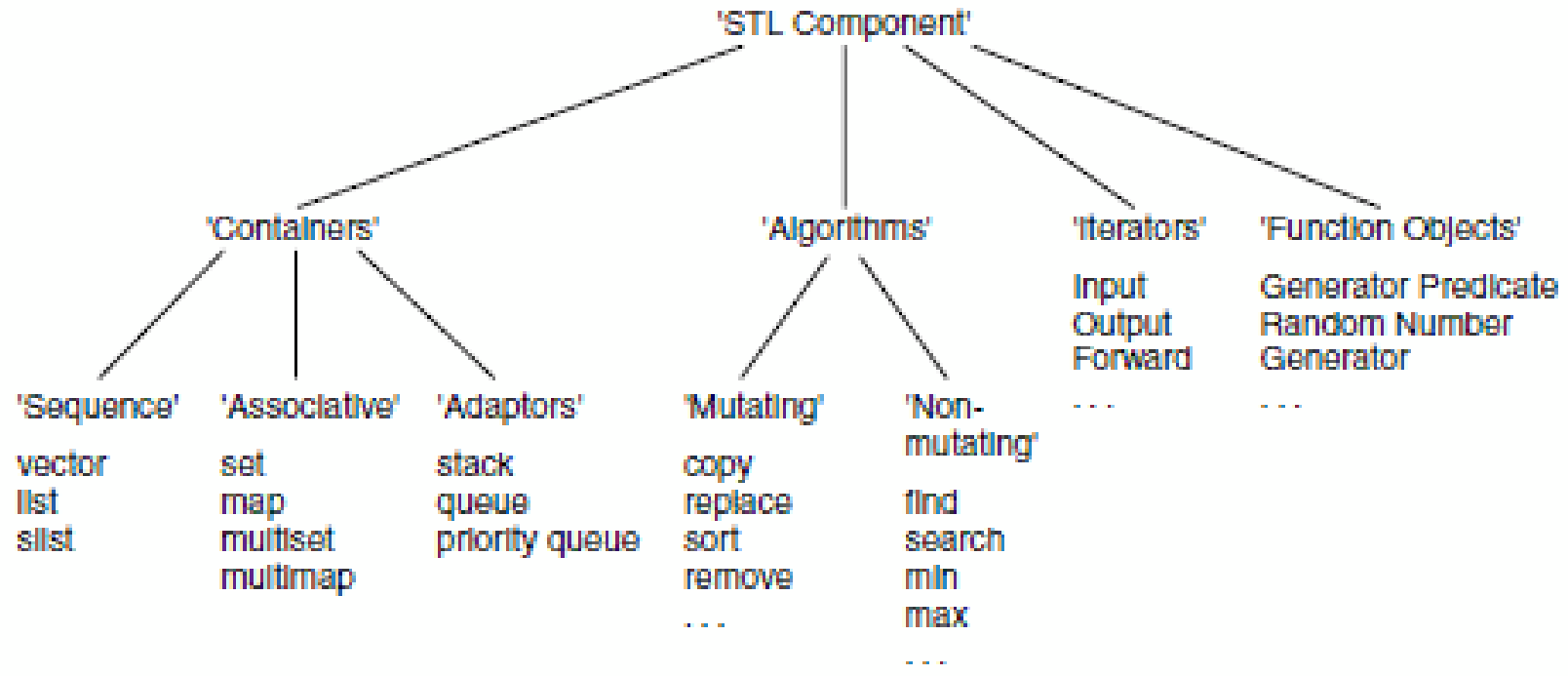


Figure 3.1 STL roadmap

STL – Standard Template Library

- Collections of useful classes for common data structures
- Ability to store objects of any type (template)
- Container – class that stores a collection of data
- STL consists of 10 container classes:
  - Sequence containers
  - Adapter containers
  - Associative containers

STL Containers

- Sequence Container
  - Stores data by position in linear order:
  - First element, second element , etc:
- Associate Container
  - Stores elements by key, such as name, social security number or part number
  - Access an element by its key which may bear no relationship to the location of the element in the container
- Adapter Container
  - Contains another container as its underlying storage structure

STL Containers

- Sequence Container
  - Vector
  - List
  - Slist
- Adapter Containers
  - Stack
  - Queue
  - Priority queue
- Associative Container
  - Set, multiset
  - Map, multimap

Vector Container

- Generalized array that stores a collection of elements of the same data type
- Vector – similar to an array
  - Vectors allow access to its elements by using an index in the range from 0 to n-1 where n is the size of the vector
- Vector vs array
  - Vector has operations that allow the collection to grow and contract dynamically at the rear of the sequence

Vector Container

Example:

```
#include <vector>

vector<int> scores (100);                    //100 integer scores
vector<Rectangle> rectangleList(20);         //list of 20 rectangles
```

Vector Container

- Allows direct access to the elements via an index operator
- Indices for the vector elements are in the range from 0 to size() -1
- Example:

```
#include <vector>
vector <int> v(20);
v[5]=15;
```

Vector

- A sequence that supports random access (direct access) to elements
  - Constant time random access (read/write)
  - Commonly used operations
    - begin(), end(), size(), [], push_back(…), pop_back(), insert(…), empty()
  - Elements can be inserted and removed at the beginning, the end and the middle

Example of vectors

```
// Instantiate a vector
vector<int> V;       // size() = 0

// Insert elements
V.push_back(2);          // v[0] == 2, 1st element
V.insert(V.begin(), 3); // V[0] == 3, V[1] == 2
                         // 2 is shifted forward,
                         // 3 is added at begin()

// Random access
V[0] = 5;          // V[0] == 5

// Test the size
int size = V.size();// size == 2, item1 is 5, item2 is 2
```

Vector Operations

See

www.cplusplus.com/reference/vector/vector/

For list of vector operations.

List Container

- Stores elements by position
- Each item in the list has both a value and a memory address (pointer) that identifies the next item in the sequence
- To access a specific data value in the list, one must start at the first position (front) and follow the pointers from element to element until data item is located.
- List is not a direct access structure
- Advantage: ability to add and remove items efficiently at any position in the sequence

STL List Class

Constructors and assignment

- list <int> aList(5, 10);
- list<int> vList(aList);
- List<int> bList;
- bList = aList;

Access

- bList.front() --- returns 1st element in the list
- bList.back() --- returns the last element in the list

Insert and Remove

- bList.push_front(20) -- Inserts a new element at the beginning of the list
- bList.pop_back() --- Removes the last element in the list

Stack Container

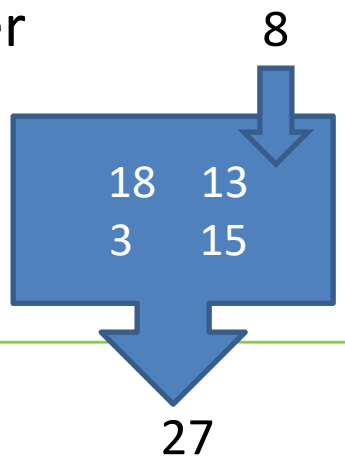
- Adapter Container
- These containers restrict how elements enter and leave a sequence
- Stack
  - allows access at only one end of the sequence (top)
  - Adds objects to container by *pushing* the object onto the stack
  - Removes objects from container by *popping* the stack
  - LIFO ordering (last end, first out)

Queue Container

- Queue
  - Allows access only at the front and rear of the sequence
  - Items enter at the rear and exit from the front
  - Example: waiting line at a grocery store
  - FIFO ordering (first-in first-out)
  - *push(add* object to a queue)
  - *pop* (remove object from queue)

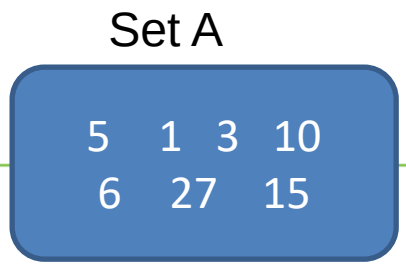
Priority Queue Container

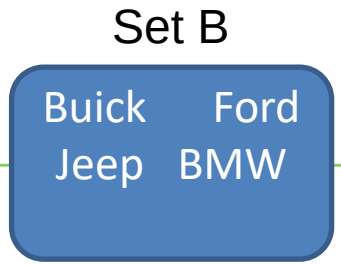
- Priority queue
  - Operations are similar to those of a stack or queue
  - Elements can enter the priority queue in any order
  - Once in the container, a delete operation removes the largest (or smallest) value
  - Example: a filtering system that takes in elements and then releases them in priority order



Set Container

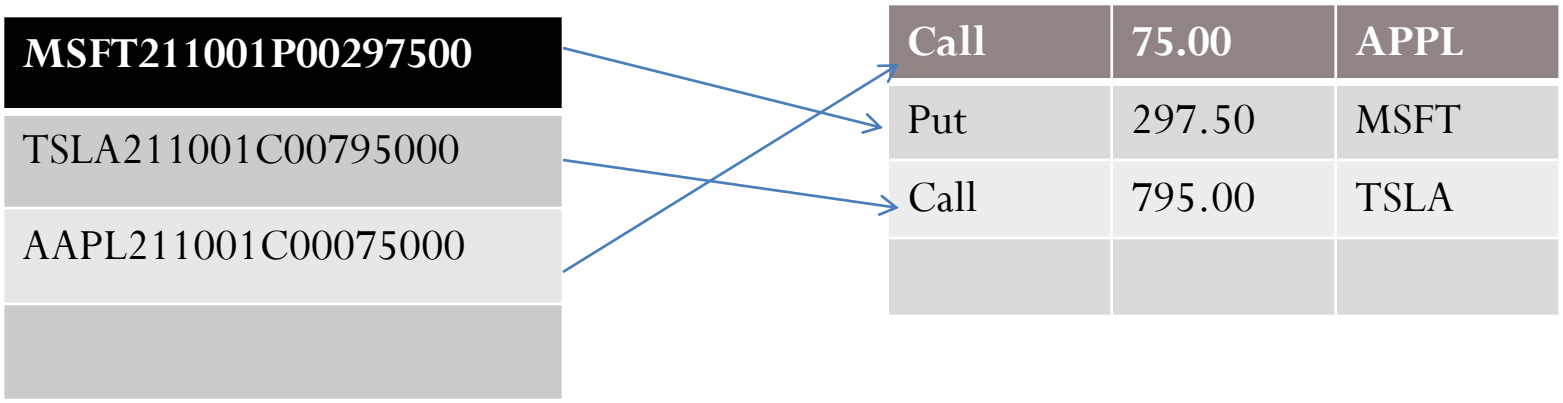
- Set
  - Collection of unique values, called keys or set members
  - Contains operations that allow a programmer to:
    - determine whether an item is a member of the set
    - insert and delete items very efficiently





Map Container

- Implements a key-value relationship
- Programmer can use a key to access corresponding values
- Example: key could be an option contract name such as MSFT211001P00297500 that corresponds to a Put option with striking price: 297.50 for symbol MSFT.



Map example

```
#include <iostream>
#include <map>
using namespace std;

struct Option
{
        string type;
        double price;
        string symbol;
        Option() : type(""), price(0), symbol("") {}
        Option(string type_, double price_, string symbol_)
                : type(type_), price(price_), symbol(symbol_) {}
};
```

Map example (Continue)

```
int main()
{
	map<string, Option> myOptions;
	myOptions["MSFT211001P00297500"] = Option("Put", 297.50, "MSFT");
	myOptions["TSLA211001C00795000"] = Option("Call", 795.00, "TSLA");
	myOptions["AAPL211001C00075000"] = Option("Call", 75.00, "APPL");

	return 0;
}
```

Multi-Set Container

- A multi-set is similar to a set, but the same value can be in the set more than once
- Multi-set container allows duplicates

Multi-map Container

- Similar to a map container
- Multi-map container allows duplicates

How to access Components - Iterator

- Iterator is an object that can access a collection of like objects one object at a time.
- An iterator can traverse the collection of objects.
- Each container class in STL has a corresponding iterator that functions appropriately for the container
- For example: an iterator in a vector class allows random access
- An iterator in a list class would not allow random access (list requires sequential access)

Common Iterator Operations

*	Return the item that the iterator currently references
++	Move the iterator to the next item in the list
--	Move the iterator to the previous item in the list
==	Compare two iterators for equality
!=	Compare two iterators for inequality

STL List Iterator

Iterator Declaration

- list<T>::iterator itr;

Iterator Options

- itr = l.begin() set iterator to beginning of the list
- Itr = l.end() set iterator to after the end of the list

See www.cplusplus.com/reference/list/list/ for list of STL list operations.

```
#include <algorithm>
#include <iostream>
#include <list>
using namespace std;
void main()
{           int myints[] = {7, 5, 16, 8};
            list<int> l (myints,myints+4);
            l.push_front(25);// Add an integer to the front of the list
            l.push_back(13);// Add an integer to the back of the list
            auto it = find(l.begin(), l.end(), 16); // Insert an integer before 16 by searching
            if (it != l.end()) { l.insert(it, 42);          }
            list<int>::iterator itr;
            for (itr = l.begin(); itr != l.end(); itr++)
                  cout << *itr << "  ";
            cout << endl;
 }
/*    25  7  5  42  16  8  13          */
```

STL Map Iterator Example

```
int main()
{     map<string, Option> myOptions;
      myOptions["MSFT211001P00297500"] = Option("Put", 297.50, "MSFT");
      myOptions["TSLA211001C00795000"] = Option("Call", 795.00, "TSLA");
      myOptions["AAPL211001C00075000"] = Option("Call", 75.00, "APPL");
      for (map<string, Option>::iterator itr = myOptions.begin(); itr != myOptions.end(); itr++)
      {         cout << itr->first << "   " << itr->second.type << "    "
                        << itr->second.price << "     " << itr->second.symbol << endl;
      }
      return 0;
}
/*
AAPL211001C00075000      Call     75       APPL
MSFT211001P00297500      Put      297.5    MSFT
TSLA211001C00795000      Call     795      TSLA
*/
```

STL Map Example

```
#include <string>
#include <iostream>
#include <map>
using namespace std;
int main()
{   map<string,int> my_map;
    my_map["x"] =  11;     my_map["y"] = 23;
    auto it = my_map.find("x");
    if (it != my_map.end()) cout << "x: " << it->second << endl;
    it = my_map.find("z");
    if (it != my_map.end()) cout << "z1: " << it->second << endl;
    // Accessing a non-existing element creates it
    if (my_map["z"] == 42) cout << "Oha!" << endl;
    it = my_map.find("z");
    if (it != my_map.end()) cout << "z2: " << it->second << endl;
    return 0;
}
```

Writing classes that work with the STL

- Classes that will be stored in STL containers should explicitly define the following:
  - Default constructor
  - Copy constructor
  - Destructor
  - operator =
  - operator==
  - operator<
- Not all of these are always necessary, but it might be easier to define them than to figure out which ones you actually need
- Many STL programming errors can be traced to omitting or improperly defining these methods

References

- Numerical Methods in Finance with C++ (Mastering Mathematical Finance), by Maciej J. Capinski and Tomasz Zastawniak, Cambridge University Press, 2012, ISBN-10: 0521177162
- *Financial Instrument Pricing Using C++, Daniel J. Duffy, ISBN 0470855096, Wiley, 2004.*
- *C++ STL, www.cs.mtsu.edu/~jhankins/files/3110/presentations/stl.ppt*
- *Inheritance in C++, cs.njit.edu/maura/ClassNotes/CIS601/lecture6.ppt*
- *Function Templates; C++ Class Templates, web.cse.ohio-state.edu/~neelam/courses/45922/Au05Somasund/PPT/Lecture7.ppt*