The Standard Template Library

The Standard Template Library

- A library of data structures.
- Built using templates.
- We will use it extend FMLib to multiple stocks.

But first we need to learn some techniques for dealing with overly long class names like

```
vector< shared_ptr<Priceable> > securities
```

typedef

```
typedef std::shared_ptr<Priceable> SPPriceable;
```

In general

```
typedef <<Complex Type>> <<Abbreviation>>;
```

The result

```
    vector<SPPriceable> securities;
```

Member types

```
class Priceable {
public:
    typedef shared_ptr<Priceable> sp;
    typedef vector< sp > spVec;
    // ... more code ...
};

class Portfolio {
private:
    Priceable::spVec securities;
    // ... more code ...
};
```

typename

You can also write

```
typename Priceable::spVec securities;
```

- Sometimes you have to do this when working with templates to get rid of compiler problems.
- If in doubt, use typename when working with templates

The purpose of member types

```
template <typename V>
typename V::value_type sumVector(const V& vector) {
    typename V::value_type total = 0;
    for (int i = 0; i < (int)vector.size(); i++) {
        total += vector[i];
    }
    return total;
}
```

Note that we are using `value_type`. This is a member type of `vector`. All *containers* have this member type.

The auto keyword

If the type can be deduced automatically, you can use `auto`.

```
double d = 4.0;
auto s = sqrt(d);
```

The type of the return values of a function can be deduced automatically.
Auto is great. Use it heavily!

auto with const and &

```
vector<double> vec(10,0.0);
auto& dRef = vec[5];
dRef = -1.0;
ASSERT(vec[5] == -1.0);

auto d = vec[6];
d = -1.0;
ASSERT(vec[6] == 0.0);

const auto& dRef2 = vec[7];
ASSERT(dRef2 == 0.0);
```

Using iterators

```
vector<double> v({ 1.0, 2.0, 3.0 });

double sum = 0.0;
vector<double>::iterator i = v.begin();
while (i != v.end()) {
    sum += *i;
    i++;
}

ASSERT(sum == 6.0);
```

Iterators

- Containers have `begin` and `end` methods that return iterators.
- Iterators have ++, * and == functions so they can by used just like pointers.
- A vector is a container. We will also see: `set`, `list`, `map` and `unordered_map`.

Using iterators to write

```
void setZero(vector<double>& v) {
    vector<double>::iterator i = v.begin();
    while (i != v.end()) {
        *i=0;
        i++;
    }
}
```

Using const iterators

```
double sumVector( const vector<double>& v ) {
    double sum = 0.0;
    vector<double>::const_iterator i = v.begin();
    while (i != v.end()) {
        sum += *i;
        i++;
    }
    return sum;
}
```

auto makes iterators more bearable

```
double sumWithAuto(const vector<double>& v) {
    double sum = 0.0;
    auto i = v.begin();
    while (i != v.end()) {
        sum += *i;
        i++;
    }
    return sum;
}
```

for loops and containers

- `Matrix` is a container.
- It has begin and end methods that return pointers. Pointers are iterators.

```
Matrix matrix("1,3;2,4");
double total = 0.0;
for (auto d : matrix) {
    total += d;
}
ASSERT_APPROX_EQUAL(total, 10.0, 0.001);
```

This special syntax can be used for all containers.

Making Matrix more of a container

To make Matrix a fully fledged container, we've added the following member typedefs.

```
    typedef double value_type;
    typedef double* iterator;
    typedef const double* const_iterator;
```

You should follow as many of the container conventions as make sense when you write a container.

A generic sum function

This function will now work with vectors and matrices.

```
template <typename C>
typename C::value_type sumContainer(const C& c) {
    typename C::value_type total = 0;
    for (auto v : c ) {
    total +=v;
   }
    return total;
}
```

The container set

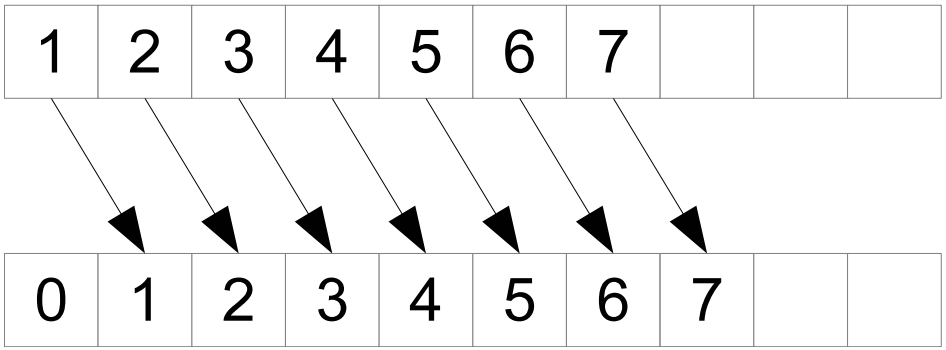
- Stores items in order without duplicates
- Items must override < to define what "in order" and "duplicate" actually mean.

```
set<int> ints;
ints.insert(1);
ints.insert(3);
ints.insert(2);
ints.insert(3); // duplicate ignored
ASSERT(ints.size() == 3);
for (auto i : ints) {
    std::cout << "Item " << i <<"\n";
}
```

Performance

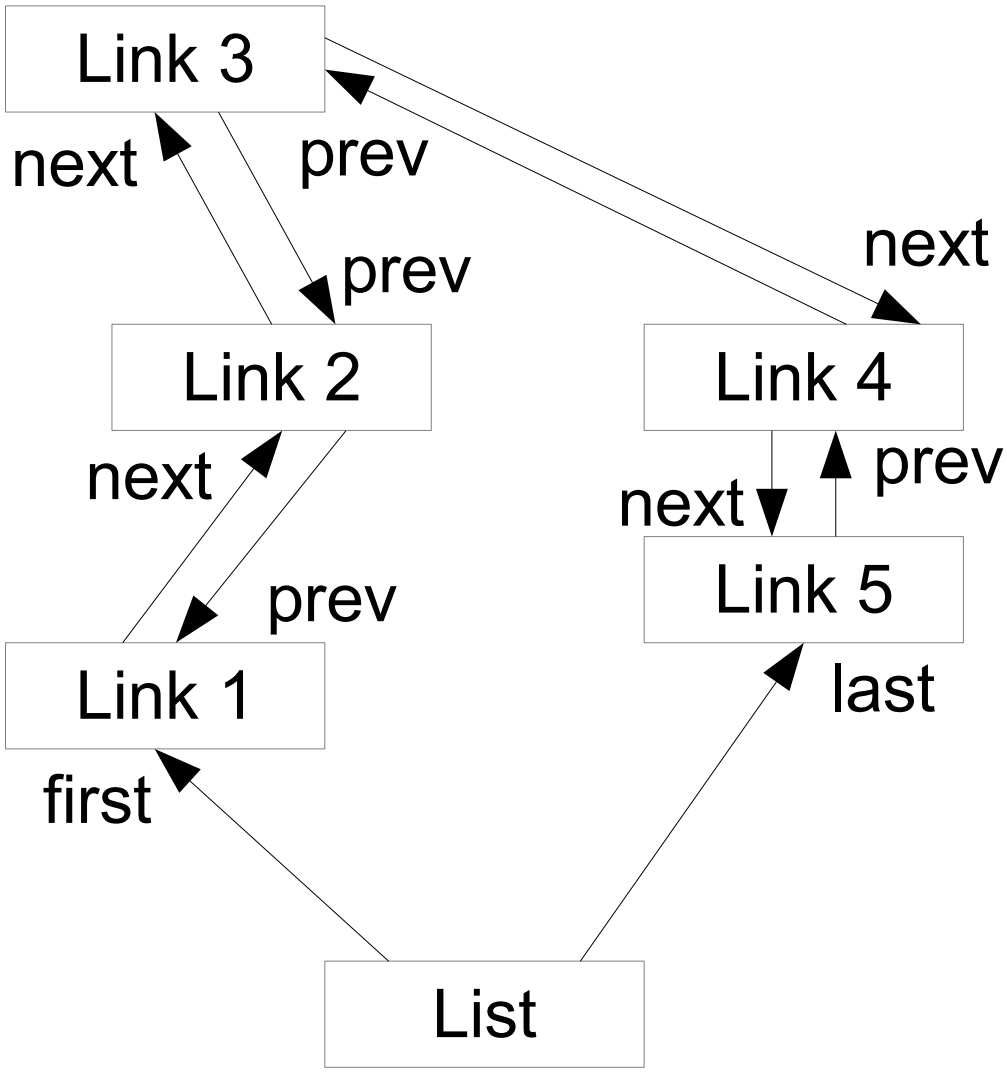
- For a set it takes $O(\log(n))$ to insert elements.
- There is also `unordered_set` which is based on a hash map which is usually quicker. Roughly $O(1)$ to insert elements.
- (n is the size of the set.)

Vector



- A vector takes $O(1)$ to find the entry at index i.
- It takes $O(n)$ to insert at the beginning.
- It allocates more memory than needed initially, so that adding at the end takes $O(1)$ normally. Takes $O(n)$ if you exceed the available capacity.

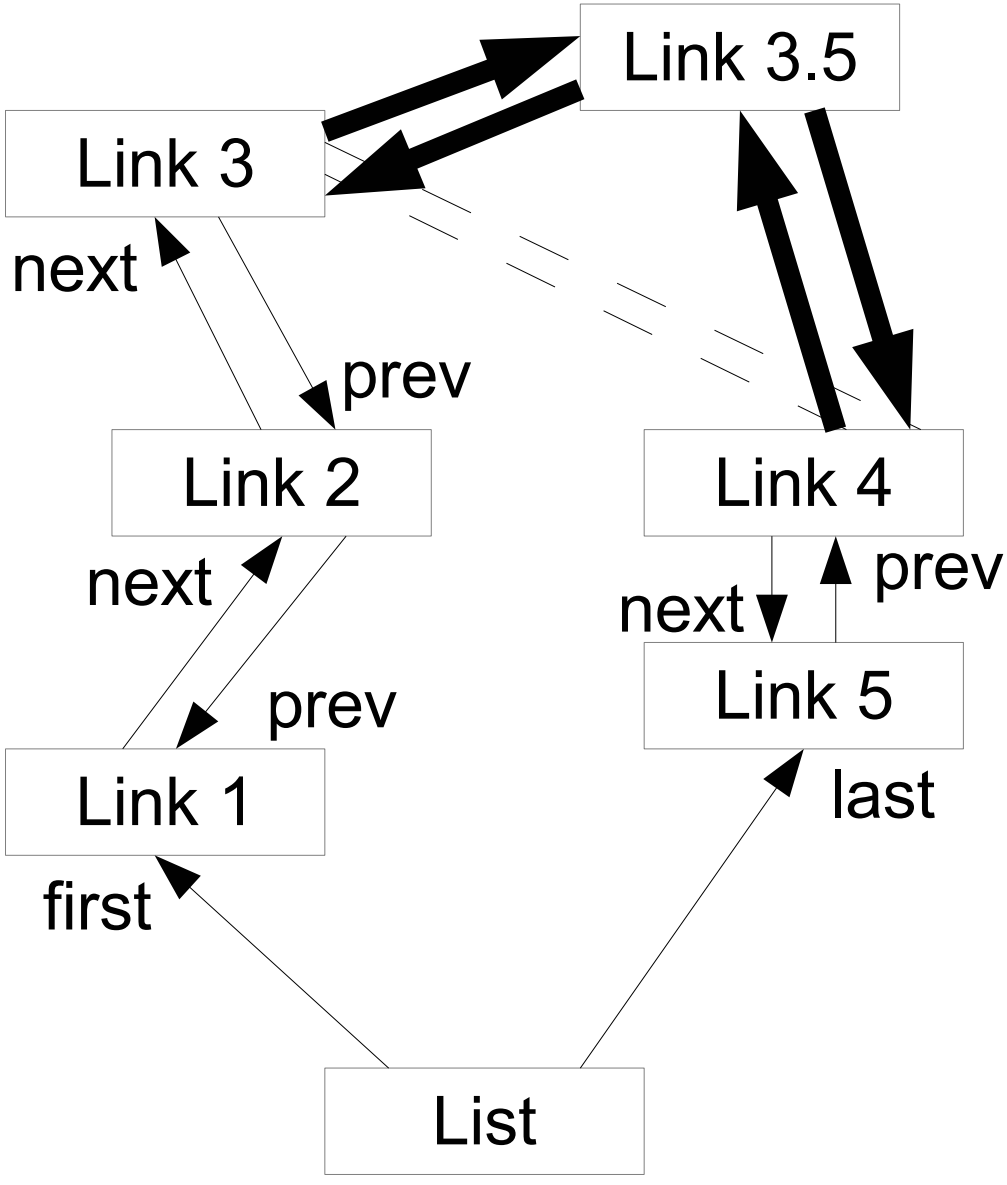
list


Link 3
next
prev
prev
next
Link 2
Link 4
next
prev
next
prev
Link 5
Link 1
last
first
List

list

```
class Link {
    Data d;
    Link* next;
    Link* previous;
};
```

Inserting in a list



List performance

- $O(1)$ to insert at the beginnging.
- $O(1)$ to insert at the end.
- $O(n)$ to find a the entry at index i.
- $O(1)$ to insert at a known link.

Using a list as a priority queue

```
// use a list to store items in priority order
list<string> list;
list.push_back("Drinking");
list.push_back("Dancing");
list.push_front("Exam");
list.push_front("Revision");

std::cout << "Todo list\n";
for (auto item : list) {
    std::cout << "Item " << item << "\n";
}
```

Result

```
Todo list
Item Revision
Item Exam
Item Drinking
Item Dancing
```

Finding and inserting

```
auto i = list.begin();
while (i != list.end()) {
    if (*i == "Exam") {
        list.insert(i, "Dentist");
        break;
    }
    i++;
}

std::cout << "Todo list\n";
for (auto item : list) {
    std::cout << "Item " << item << "\n";
}
```

Using algorithm

- The `find` function is the best way to find an entry.

```
auto iter = find(list.begin(), list.end(), "Exam");
list.insert(iter, "Dentist");
```

- Note that you can use find to search any region of your container by providing iterators to the start and end.
- The `algorithm` library is full of useful functions like this that all work in similar ways.

The container initializer_list

```
std::initializer_list<double>
    list = { 1, 2, 3, 4 };
for (auto d : list) {
    std::cout << "Value "<<d<<"\n";
}
```

Very useful for writing tests and constructors.
Most containers have a constructor that takes an `initializer_list`.

```
std::vector<double> v({ 1, 2, 3, 4 });
```

Dictionary like containers

- `map` stores mappings from a key to some value.
- A dictionary stores mappings from a word to its meaning.
- A book's index stores mappings from words to page numbers.
- A phone book stores mappings from names to phone numbers.

The container map

```
map<string, string> fruitToCol;
fruitToCol["apples"] = "green";
fruitToCol["bananas"] = "yellow";
fruitToCol["plums"] = "purple";
fruitToCol["oranges"] = "orange";
for (const pair<string, string>& p : fruitToCol) {
    cout << "The color of ";
    cout << p.first;
    cout << " is ";
    cout << p.second;
    cout << "\n";
}
```

Looking up items

With maps, looking up an item is fast.

```
auto i = fruitToCol.find("plums");
cout << "Plums are " << (i->second)<<"\n";
```

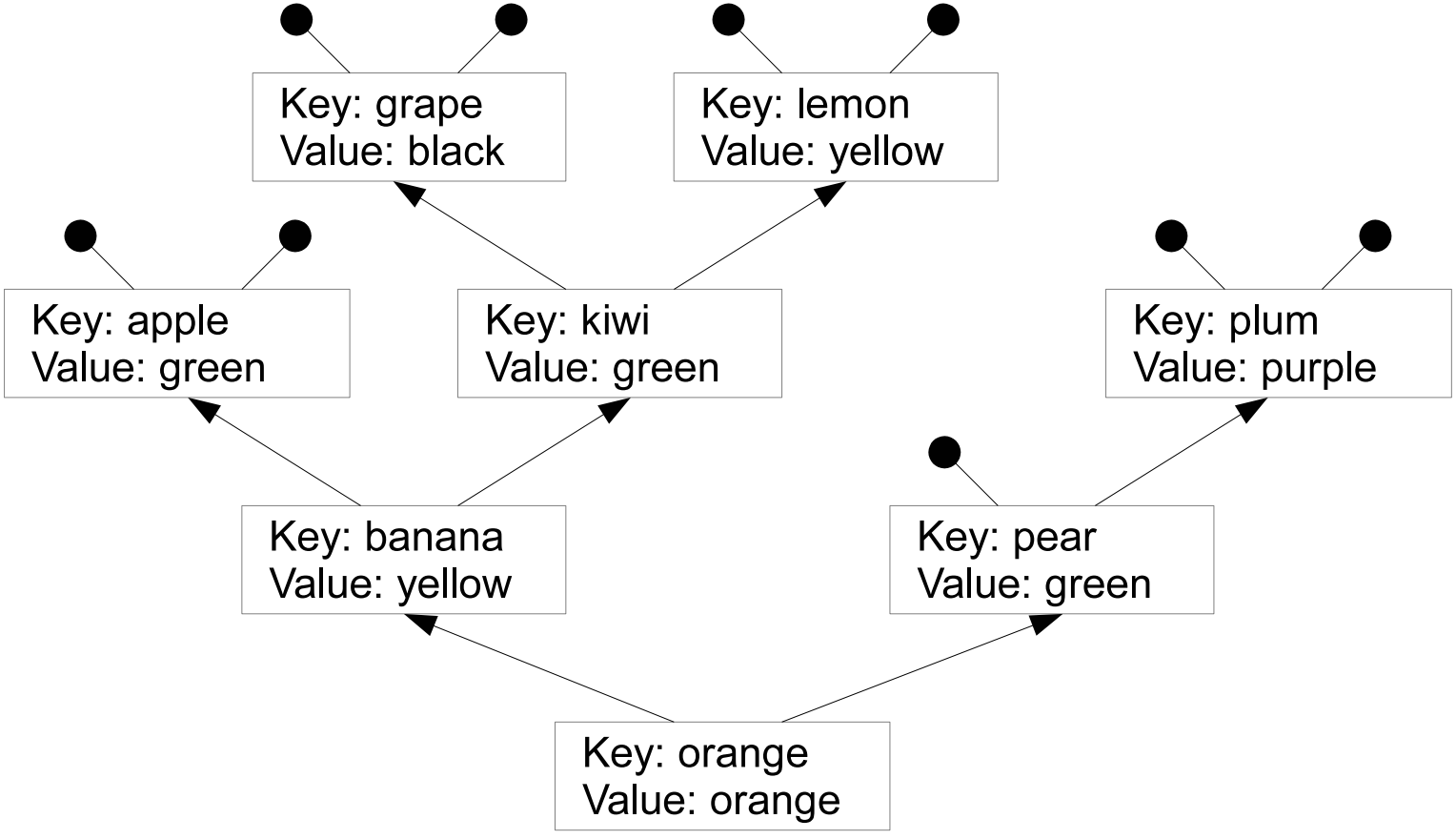
Dealing with missing items:

```
string fruit = "jackfruit";
auto iter = fruitToCol.find(fruit);
if (iter == fruitToCol.end()) {
    cout << "The color of " << fruit;
    cout << " is unknown\n";
} else {
    cout << fruit << " are " << (i->second) << "\n";
}
```

map and unordered_map

- `map` stores data ordered by key. So the keys must have a < function.
- `unordered_map` stores data using a hash algorithm (see later).
- Both classes are used in almost identical ways
- `unordered_map` is normally faster.

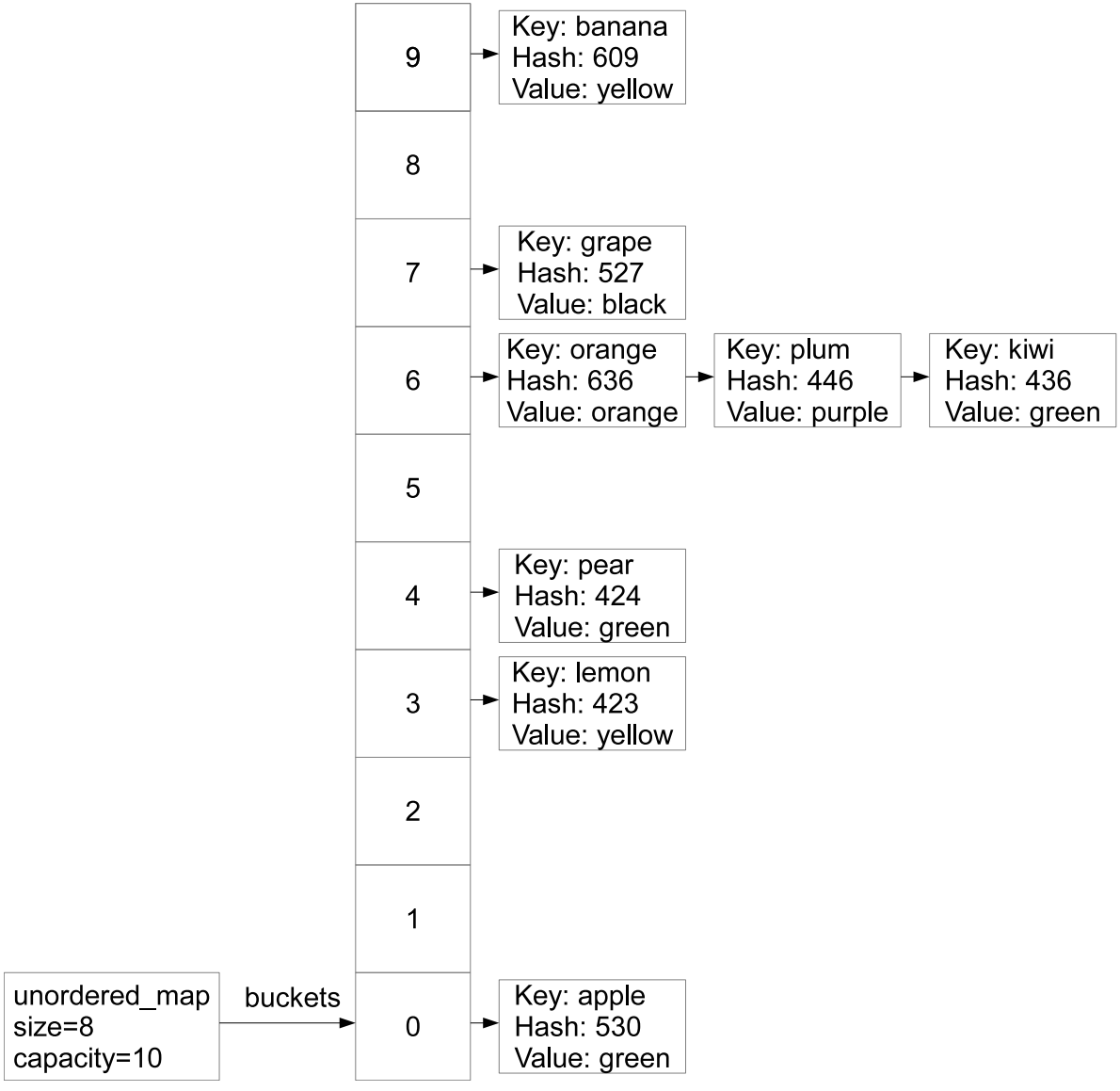
Under the covers of map



Performance of map

- Lookup takes $O(\log n)$
- Insert takes $O(\log n)$

Under the covers of a hash map



Performance of map

- Lookup takes roughly $O(1)$.
- Insert takes roughly $O(1)$ unless we need to rehash.
- You need to choose a good hash code.

Storing data objects in containers

- Data is stored by value in containers. This means it is copied whenever you add items.
- Very often you should store objects in containers using `shared_ptr` references. This is exactly what we did with the Portfolio class earlier.

A multi stock model

- z_t is a vector of n log stock prices.
- ϵ_t is a vector of R risk factors.
- A is matrix showing how the risk factors effect stock price.
- η_t is an n vector determining the trend of the risk factors.

$$\underline{z}_{t+\delta t} = \underline{z}_t + \underline{\eta}\delta t + (\delta t)^{\frac{1}{2}} A\underline{\epsilon}_t \tag{1}$$

Linear algebra

- The covariance matrix is $(\delta t)AA^T$.
- Write $\Omega = AA^T$ for the covariance matrix over a year.
- Typically we measure Ω from market and then find A satisfying the above.
- A is called a pseudo-square root of Ω.
- One algorithm to find an $n \times n$ pseudo-square root is called Cholesky decomposition. Works assuming Ω is symmetric and positiv-definite. In this case $n = R$ and we have as many risk factors as stocks.
- To speed up processing ,you may use principle-component analysis to find an approximate pseudo-square root with less risk factors.

Q-measure model

- If we ignore all the other stocks and just compute what happens to the i-th stock, it follows the Black–Scholes model.
- The mean of the growth in the log of the i-th stock price is given by:

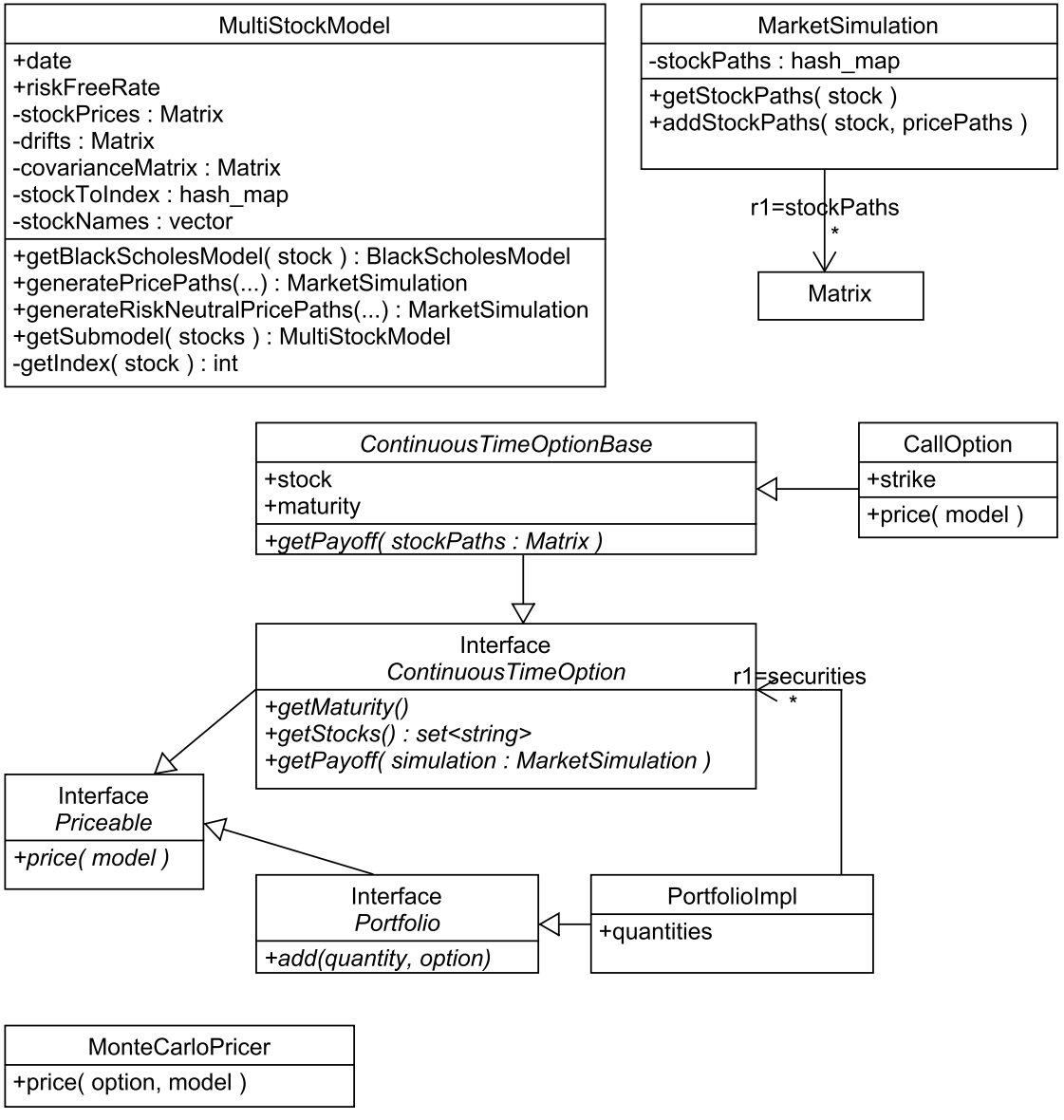
$$\underline{\eta}_{(i)}$$

- This must be equal to :

$$r - \frac{1}{2}\Omega_{(i,i)}$$

in a Q measure model.

Changing FMLib for multiple stocks



The link with data structures

- `MultiStockModel` stores a map from the name of a stock to its index in our mathematical model.

```
int getIndex(const std::string&  stockCode)
        const {
    auto pos = stockToIndex.find(stockCode);
    ASSERT(pos != stockToIndex.end());
    int idx = pos->second;
    return idx;
}
```

The MarketSimulation class

- The market simulation class stores a map from a stock name to the simulations for that stock

```
SPCMatrix getStockPaths( const std::string& stock)
    const {
    auto pos = stockPaths.find(stock);
    ASSERT(pos != stockPaths.end());
    return pos->second;
}
```

Note that SPCMatrix is a typedef for a shared pointer to a const matrix.
These give a few examples of how containers are used in this more sophisticated model.

Summary

- `typedef` keyword allows us to abbreviate complex type names.
- The `auto` keyword allows you to avoid typing the full name of a class.
- Classes can contain member types.
- A container is any class that stores data and returns iterators when you call `begin` and `end`.
- There is a special syntax for looping through containers using `for`.
- C++ contains numerous container classes that make it easy to store data. They have different performance characteristics.
- The library `<algorithm>` contains a number of functions that are very useful for working with containers, such as `find`.
- You should not store large objects in containers. Store them by reference using `shared_ptr` instead.