IB9N7 C++ for Quantitative Finance

Lecture 14: STL

J.F.Reizenstein@warwick.ac.uk

18 February 2016
(Week 20)

Outline

In this lecture, we'll look in more detail at some of the basic components of the Standard Template Library (STL):

- containers;
- iterators;
- some details of sorting.

Note that STL is not entirely OOP based.

It has its own style, which is a combination of generic programming, objects, functional style and data abstraction.

There are good reasons for this!

First look

What is the Standard Template Library?

Standard Template Library

The **Standard Template Library (STL)** is a collection of templated containers and associated algorithms for manipulating those containers.

- A **container** is a collection of objects of an arbitrary type (including fundamental types).
- **Iterators** are used to iterate over the elements inside these containers in a somehow safe and generalised manner (c.f. pointers).
- **Algorithms** allow sorting, searching, and other manipulations of these containers, generally operating through iterators.
- Each container type is a **C++ template class**.

Templates

Where a particular function or class would have the same or similar implementation for different types, it is not always necessary to provide a separate implementation or overload for each of those types.

One can instead provide a generic implementation in terms of generic types, and C++ can assume that the function or class was defined for the specific types that you attempt to use them with.

This is how the `std::vector` class is defined in the standard library.

- There is no explicit class `std::vector<double>`, but there is a `std::vector<T>` defined for a generic type `T`.

Motivating example

You might reasonably define a function like the following:

Example: Sum function

```
double calculate_sum (const std::vector<double> &
    data)
{
   double sum = 0.0;
   for (std::vector<double>::size_type i = 0u; i !=
       data.size(); ++i)
      sum += data.at(i);
   return sum;
}
```

But this is an example of something much more generic. . .

Motivating example: templatised

Here is a more generic version:

Example: Sum function

```
template <class T>
T calculate_sum (const std::vector<T> & data, T
    starting_value)
{
   T sum = starting_value;
   for (typename std::vector<T>::size_type i = 0u; i
       != data.size(); ++i)
      sum += data.at(i);
   return sum;
}
```

STL vectors

We have already met STL vectors.

Example

```
std::vector<double> vec(100u);
```

creates a vector of size 100 with elements of type `double`.

Vectors support indexed access of their elements, e.g.:

```
for (std::vector<double>::size_type i = 0u; i != vec.
    size(); ++i)
{
   std::cout << vec.at(i) << std::endl;
}
```

First look: iterators

There are other containers besides vectors.
Not all containers can be iterated over in the above way.
However, all containers provide iterators in the following form:

Example

```
for (std::vector<double>::iterator i = vec.begin(); i
    != vec.end(); ++i)
   std::cout << *i << std::endl;
```

The above code outputs each of the vector's elements on the standard output.

Iterators are a(nother) generalisation of pointers, and help to avoid off-by-one errors. . .

as long as you remember that `vec.end()` cannot be dereferenced.

First look: algorithms

STL also provides algorithms for common tasks.

For example, to fill a vector with a constant, we don't need to loop over it ourselves. Instead, we can do:

Filling a vector

```
std::fill(vec.begin(), vec.end(), 1.0);
```

Note: algorithms are usually defined as **free functions** acting on iterators, rather than as member functions of the containers. This is to avoid duplication.

Containers

STL containers

STL containers fit into two categories:

Sequential containers

Programmer controls the order in which the elements are stored and accessed.
All sequential containers support sequential access.
Important examples: `std::vector`, `std::list`.

Associative containers

Associative containers associate **keys** with **values** (in some cases, the **value** also acts as the **key**).
Associative containers implement data structures that can be quickly searched.
Important examples: `std::map`, `std::set`.

Selection of containers

Each container has:

- specific rules governing the relationship between the values in the container. e.g. uniqueness, sorting, access.
- an appropriate underlying data structure.
- a slightly differing set of routines depending on the nature of the container.
  Some parts of the interface are shared.

The choice of container depends on the required functionality and the efficiency expectations for the application.

STL vectors

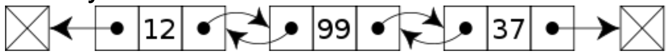
`std::vector`

- Requires header: `<vector>`
- The only C++ container with contiguous storage.
- Supports fast random access.
- Inserting or deleting elements other than at the back may be slow.
- Adding/removing elements does not always result in the underlying array being reallocated (the current size of the underlying array is known as the **capacity** and is distinct from the current number of elements).
- This is the 'default' container of C++ and is a good choice if you are unsure which to use.

STL lists

`std::list`

- Requires header: `<list>`
- Doubly linked list.



- Supports only bidirectional sequential access (no random access).
- Constant time (but not necessarily fast) insertion and deletion at any point.
- Best suited when frequent deletion or insertion of elements at random points are needed, and walking forward/backward is more common than needing to know the element in the middle.

Creating vectors and lists

`std::vector` and `std::list` provide:

- default constructors that create empty an vector or list, respectively.
- the usual copy constructor.
- constructors that allow specification of the initial number of elements, and with a specified initial value.
  If no initialising value is specified, default value-initialisation is used.

One specifies the datatype of the elements of the vector/list in the usual way for template classes. We have seen this by example:

```
std::vector<double> vec1;
std::list<double> list1;
```

Accessing elements

One can access (and assign) elements of `std::vector` through the `at` member function:

```cpp
std::vector<double> vec(10u);
vec.at(2u) = 3.0;
```

Note that `std::list` does not provide random access and thus no `at` function. More general element access (e.g. looping through all elements) is possible through iterators.

Iterators

Iterating over a vector

With vectors being so similar to arrays, we can use a standard `for`-loop technique to iterate over the elements:

Example: classical iteration over a vector

```cpp
std::vector<int> vec(10u);
for (std::vector<int>::size_type i = 0u; i != vec.size();
    ++i)
   vec.at(i) = rand();
```

Here, 10 is a magic number, but it is only used once.

- It is not used again in the `for` loop condition.
- The `size` member function is used instead.

However, this approach is not always possible with the other containers in C++.

- Instead, we should use the more abstract **iterator** mechanism.

Iterators

Iterator

An iterator marks a position in a container object and allows such a container to be traversed.

- To be an iterator, an entity needs to support at least ++ (pre and post incrementing), * (dereferencing for either input or output).
- In many senses, iterators are similar to pointers (in fact, pointers are valid iterators) due to pointer arithmetic.
- Each STL container provides iterators for accessing its contents.
- An iterator type is specific to the type of container it is associated with, and is defined as a member type of the container.
- An iterator is specific to the container it is associated with.

Iterators

An iterator for a `std::vector<double>` would be declared as:

```cpp
std::vector<double>::iterator iter;
```

There are several types of iterators (discussed later).

At a minimum, iterators can be incremented to point to the next element in the container, eg:

```cpp
++iter;
```

If they point to valid elements, they can also be dereferenced to access the value at the position they mark (to read and/or write, depending on type), again analogously to pointers:

```cpp
std::cout << *iter << std::endl; // read dereference
```

Iterating over a vector using an iterator

Example: Iterating over a vector

```cpp
std::vector<int> vec(10u);
for (std::vector<int>::iterator i = vec.begin(); i !=
    vec.end(); ++i)
      *i = rand();
```

- The iterator `i` here is specific to the vector `vec` and could not be used with a different container, type of container or different element datatype.
- The `begin` and `end` member functions of a container return iterators marking the beginning and **one beyond the end** of the container, respectively.

Invalidated iterators

Any operations on a container that changes the size of the container may invalidate any iterators obtained previously.

Refer to your favourite reference for the full rules on when/which iterators get invalidated.

Example: iterator invalidation

```cpp
std::vector<double> vec(10u);
std::vector<double>::iterator iter = vec.begin();
vec.resize(20u); // resizing a vector may invalidate all its iterators
std::cout << *iter << std::endl; // Oops! iter may no longer
    be valid
```

This is much like trying to dereference a pointer that points to a no-longer-existent object.

Inserting into containers

To insert a value into the middle of a container:

```cpp
vec.insert(iter, 7.0);
```

- `iter` is an iterator specifying the position in the container which the value is to be inserted **before**. Note that it is easy to remember this interface inserts **before** the existing element, because it is the only way it can work due to the valid range of iterators. To insert before the first element, use the `begin` iterator, and to insert after the last element (**before** the (last+1) sentinel), use the `end` iterator.
- In this case, we will insert the value 7.0.

Removing elements from a container

(Invalidates iterators.)
Removing elements from container is similar to inserting, by simply providing an iterator to the element which is to be removed:

```
vec.erase(iter);
```

- After this operation, iterators on `vec` may be invalidated.
- A container can be cleared (emptied) completely:

```
vec.clear();
```

Sorting

Strict weak order

- Many parts of STL are designed to deal with objects which are capable of being sorted. They have a comparison function, by default the $<$ operator is used but the user can provide a function too.
- Alternatively, a less-than function taking two elements and returning `bool` can be provided. It could be a function name, a `std::function`, a lambda or an object with an overloaded `operator()`.
- It must be a "strict weak order", i.e.
  - Be a strict partial order, i.e. irreflexive ($a < a$) and transitive ($a < b$ and $b < c$ imply $a < c$)
  - Define “a and b are equivalent” to mean “neither $a < b$ nor $b < a$”. This is an equivalence relation.
- Using a function which is not a strict weak order where one is expected is undefined behaviour.

Algorithms and iterators

Most STL algorithms require inclusion of the `<algorithm>` header. In general, algorithms do not operate on containers directly.

Instead, they traverse through a range of elements delimited by a pair of iterators, say `first` and `last`.

They need not be the `begin` and `end` iterators of a particular container.

Just as `begin` and `end`, the pair of iterators form a half open interval, with `first` included and `last` excluded.

Therefore, to traverse through all elements of a `std::vector`, one applies algorithms to the range delimited by the `begin()` and `end()` member functions.

sort

- To sort the elements of a vector, or part of one, we have seen this already

```
std::sort(a.begin(),a.end());
```

  to get **a** to be sorted in ascending order.
- Elements will have switched positions, usually by calling a **swap** function. This is a bit like assigning them to each other, but may be more efficient, for example if **a** contains vectors. Thus any existing iterators to **a** will point to the same positions, but new values.
- **std::list** has its own **sort** function, which moves all the elements by changing the pointers. So iterators keep their values but not their positions.
- The order of equivalent elements after a call to **sort** is unspecified. **stable_sort** is like **sort** but preserves

quantiles

- If all we want to know is which element will be in a certain position after a call to **sort**, we don't need to do a full **sort**.
- C++ provides a function **nth_element** which allows us to supply an iterator to a position we are interested in.
- After the following code, **i** will point to the (possible) element it would point to if a sort were performed, with lower elements all before it and higher elements after it.

```
auto i = a.begin()+;
std::nth_element(a.begin(),i,a.end());
```

- There is also a **partial_sort**, which may be useful e.g. to get a top-10 list.

reverse order

- If we want to act on a vector in reverse order, we can always use **rbegin** and **rend** instead of **begin** and **end**.
- We can reverse the elements of a vector (no less-than needed!) with

```
std::reverse(a.begin(),a.end());
```

- Similarly, **std::list** has its own **reverse** function, just like for **sort**.

Sorted containers

- It is very common to want to maintain a sorted container, and there are certain efficient operations which can be performed on one.
- The most important is **lower_bound**, which returns an iterator to the lowest point in a sorted container where a given value could be inserted while keeping the container sorted. For example, the following pattern is often used to add a value to a sorted container.

```
a.insert(std::lower_bound(a.begin(),a.end(),
    newValue),newValue);
```

- Note that for a long vector, **lower_bound** could be much more efficient than iterating through.

Sorted containers 2

- To check if a sorted container contains an element equivalent to a given value, and find it if so, we might do this.

```
auto i = std::lower_bound(a.begin(),a.end(),value)
    ;
bool contains = (i!=a.end() && !(value < *i) ); //if
    true, i is the location.
```

Lab objectives

- Try out STL!