mobica

TRAINING CENTRE

C++ Intermediate

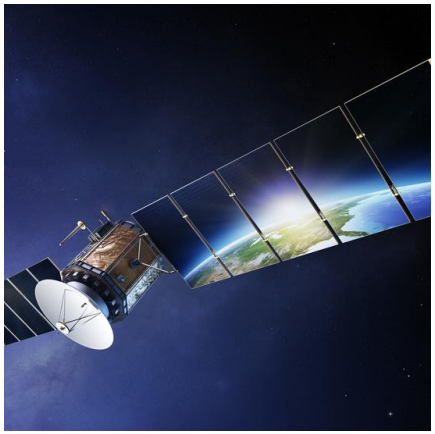
Module 7

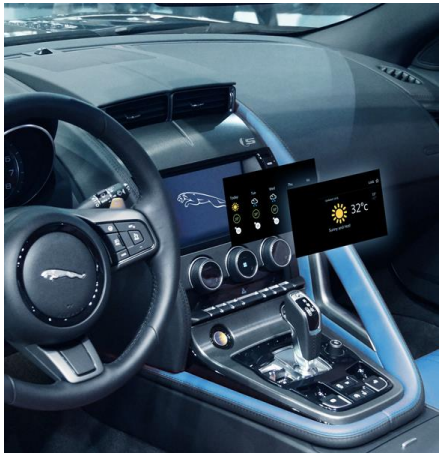
STL Containers

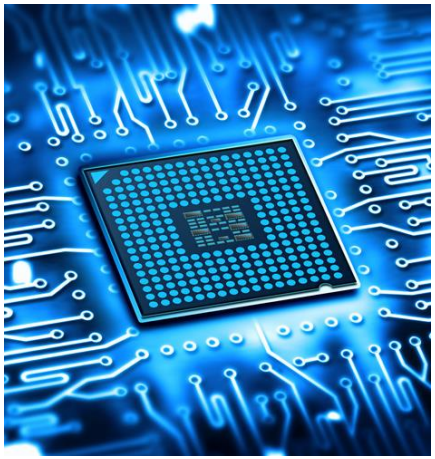
Krzysztof Wilk

October 2016

Table of Contents







Initial Requirements

Initial requirements – knowledge/skills to start this training:

Basic knowledge of:

- The differences between sequential and associative containers
- Basics of sequential and associative containers

Objectives / Expected output

Expected outcome – what knowledge will you gain after finishing this training:

After the workshop participants will have knowledge about:

- Implementation of containers regarding to memory
- Able to choose a proper container

Containers - overview

Containers - overview

A **container** is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

The container manages the **storage space** for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on **the efficiency of some of its members (complexity)**. This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.

Containers - overview

Sequence containers:

- array (C++11)
- vector
- deque
- forward_list (C++11)
- list

Container adaptors:

- stack
- queue
- priority_queue

Containers - overview

Associative containers:

- set
- multiset
- map
- multimap

Unordered associative containers:

- unordered_set (C++11)
- unordered_multiset
- unordered_map (C++11)
- unordered_multimap

Sequencial Containers

Array, Vector, Deque, Foward List and List

Sequencial Containers - array

Arrays are **fixed-size** sequence containers: they hold a specific number of elements ordered in a **strict linear sequence**.

Internally, an array does **not keep any data other than the elements** it contains (**not even its size, which is a template parameter, fixed on compile time**). It is as efficient in terms of storage size as an ordinary array declared with the language's bracket syntax ([]). This class merely adds a layer of member and global functions to it, so that arrays can be used as standard containers.

Unlike the other standard containers, arrays have a fixed size and do not manage the allocation of its elements through an allocator: they are an aggregate type encapsulating a fixed-size array of elements. Therefore, they **cannot be expanded or contracted dynamically** (see vector for a similar container that can be expanded).

Sequencial Containers - array

Zero-sized arrays are valid, but they should not be dereferenced (members front, back, and data).

Unlike with the other containers in the Standard Library, **swapping two array containers is a linear operation** that involves swapping all the elements in the ranges individually, which generally is a considerably **less efficient operation**. On the other side, this allows the iterators to elements in both containers to keep their original container association.

Another unique feature of array containers is that they can be treated as tuple objects: The <array> header overloads the get function to access the elements of the array as if it was a tuple, as well as specialized tuple_size and tuple_element types.

Sequencial Containers - array

Member functions
Iterators
begin
end
rbegin
rend
cbegin
cend
crbegin
crend

Capacity
size
max_size
empty

Element access
operator[]
at
front
back
data

Modifiers
fill
swap

Non-member function overloads
get (array)
relational operators (array)

Non-member class specializations
tuple_element<array>
tuple_size<array>

Sequencial Containers - array

```
#include <iostream>
#include <array>

int main ()
{
  std::array<int,10> myarray;

  // assign some values:
  for (int i=0; i<10; i++) myarray[i] = i+1;

  // print content:
  std::cout << "myarray contains:";
  for (int i=0; i<10; i++)
    std::cout << ' ' << myarray.at(i);
  std::cout << '\n';

  return 0;
}
```

Sequencial Containers - vector

Vectors are sequence containers representing arrays that **can change in size.**

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays**, their size can change dynamically,** with their storage being handled automatically by the container.

Sequencial Containers - vector

Internally, vectors use a **dynamically allocated array to store their elements.** This array may need to be **reallocated** in order to **grow** in size when new elements are inserted, which implies **allocating a new** array and **moving** all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some **extra storage** to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size).

Sequencial Containers - vector

Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with amortized constant time complexity (see push_back).

Therefore, compared to arrays, **vectors consume more memory** in exchange for the ability to manage storage and grow dynamically in an efficient way.

Sequencial Containers - vector

Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very **efficient accessing its elements** (just like arrays) and relatively efficient **adding or removing elements from its end**. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists.

Sequencial Containers - vector

Member functions
- (constructor)
- (destructor)
- operator=

Iterators:
- begin
- end
- rbegin
- rend
- cbegin
- cend
- crbegin
- crend

Capacity:
- size
- max_size
- resize
- capacity
- empty
- reserve
- shrink_to_fit

Element access:
- operator[]
- at
- front
- back
- data

Sequencial Containers - vector

Modifiers:

- assign
- push_back
- pop_back
- insert
- erase
- swap
- clear
- emplace
- emplace_back

Allocator:

- get_allocator

Non-member function overloads

- relational operators
- swap

Template specializations

- vector<bool>

Sequencial Containers - vector

```
  std::vector<int> myvector (10);   // 10 zero-initialized elements

  std::vector<int>::size_type sz = myvector.size();

  for (unsigned i=0; i<sz; i++) myvector[i]=i;

  for (unsigned i=0; i<sz/2; i++)
  {
    int temp = myvector[sz-1-i];
    myvector[sz-1-i]=myvector[i];
    myvector[i]=temp;
  }

  std::cout << "myvector contains:";
  for (unsigned i=0; i<sz; i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

//myvector contains: 9 8 7 6 5 4 3 2 1 0
```

Sequencial Containers - deque

Double ended queue

deque (usually pronounced like "deck") is an irregular acronym of double-ended queue. Double-ended queues are sequence containers with **dynamic sizes** that can be expanded or contracted on **both ends** (either its front or its back).

Specific libraries may implement deques in different ways, generally as some form of **dynamic array.** But in any case, they allow for the individual elements to be accessed directly through **random access iterators**, with storage handled automatically by expanding and contracting the container as needed.

Therefore, they provide a functionality similar to vectors, but with **efficient** insertion and deletion of elements also at the **beginning** of the sequence, and **not only at its end**. But, unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a deque by **offsetting a pointer to another element causes undefined behavior.**

Sequencial Containers - deque

Both vectors and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a **deque can be scattered in different chunks of storage**, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, deques are a little **more complex internally than vectors**, but this allows them to **grow more efficiently under certain circumstances**, especially with very long sequences, where **reallocations become more expensive**.

For operations that involve frequent insertion or removals of elements at positions **other than the beginning or the end, deques perform worse** and have less consistent iterators and references than lists and forward lists.

Sequencial Containers - deque

```
 std::deque<int> mydeque (10);   // 10 zero-initialized
elements
  std::deque<int>::size_type sz = mydeque.size();

  for (unsigned i=0; i<sz; i++) mydeque[i]=i;

  for (unsigned i=0; i<sz/2; i++)
  {
    int temp;
    temp = mydeque[sz-1-i];
    mydeque[sz-1-i]=mydeque[i];
    mydeque[i]=temp;
  }

  std::cout << "mydeque contains:";
  for (unsigned i=0; i<sz; i++)
    std::cout << ' ' << mydeque[i];
  std::cout << '\n';

//mydeque contains: 9 8 7 6 5 4 3 2 1 0
```

Sequencial Containers – forward_list

Forward lists are sequence containers that allow **constant time insert and erase** operations **anywhere** within the sequence.

Forward lists are implemented as **singly-linked lists**; Singly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept by the **association to each element of a link to the next element in the sequence**.

The main design difference between a forward_list container and a list container is that the first keeps internally **only a link to the next element**, while the latter keeps two links per element: one pointing to the next element and one to the preceding one, allowing efficient iteration in both directions, but consuming additional storage per element and with a slight higher time overhead inserting and removing elements. forward_list objects are thus more efficient than list objects, although they can only be iterated forwards.

Sequencial Containers – forward_list

Compared to other base standard sequence containers (array, vector and deque), forward_list perform generally better in inserting, extracting and moving elements in any position within the container, and therefore also in algorithms that make intensive use of these, like sorting algorithms.

The main drawback of forward_lists and lists compared to these other sequence containers is that they **lack direct access to the elements by their position;**
For example, to access the sixth element in a forward_list one has to iterate from the beginning to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

Sequencial Containers – list

Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.

List containers are implemented as **doubly-linked lists**; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

They are very **similar to forward_list**: The main difference being that forward_list objects are single-linked lists, and thus they can only be iterated forwards, in exchange for being somewhat smaller and more efficient.

Sequencial Containers – list

Compared to other base standard sequence containers (array, vector and deque), lists perform **generally better in inserting**, **extracting and moving elements in any position within the container for which an iterator has already been obtained**, and therefore also in algorithms that make intensive use of these, like sorting algorithms.

The main drawback of lists and forward_lists compared to these other sequence containers is that they lack direct **access** to the elements by their position; For example, to access the sixth element in a list, one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

Sequencial Containers – list

```
  std::list<int> mylist;
  std::list<int>::iterator it;

  for (int i=1; i<=5; ++i) mylist.push_back(i); // 1 2 3 4 5

  it = mylist.begin();
  ++it;       // it points now to number 2           ^

  mylist.insert (it,10);                          // 1 10 2 3 4 5

  // "it" still points to number 2                      ^
  mylist.insert (it,2,20);                        // 1 10 20 20 2 3 4 5

  --it;       // it points now to the second 20          ^

  std::vector<int> myvector (2,30);
  mylist.insert (it,myvector.begin(),myvector.end());
                                                  // 1 10 20 30 30 20 2 3 4 5
                                                  //               ^
  std::cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';


//mylist contains: 1 10 20 30 30 20 2 3 4 5
```

Associative Containers

Set, Multiset, Map and Multimap.

Associative Containers – set

Sets are containers that store **unique elements** following a **specific order**.

In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique. **The value of the elements in a set cannot be modified** once in the container (the elements are always const), but they can be **inserted or removed** from the container.

Internally, the elements in a set are always **sorted** following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).

set containers are generally **slower** than **unordered_set** containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.

Sets are typically implemented as **binary search trees**.

Associative Containers – set

```
  std::set<int> myset;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;

  // set some initial values:
  for (int i=1; i<=5; ++i) myset.insert(i*10);    // set: 10 20 30 40 50

  ret = myset.insert(20);                  // no new element inserted

  if (ret.second==false) it=ret.first;  // "it" now points to element 20

  myset.insert (it,25);                  // max efficiency inserting
  myset.insert (it,24);                  // max efficiency inserting
  myset.insert (it,26);                  // no max efficiency inserting

  int myints[]= {5,10,15};               // 10 already in set, not inserted
  myset.insert (myints,myints+3);

  std::cout << "myset contains:";
  for (it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

//myset contains: 5 10 15 20 24 25 26 30 40 50
```

Associative Containers – multiset

Multisets are containers that store elements following a specific order, and where **multiple elements can have equivalent values**.

In a multiset, the value of an element also identifies it (the value is itself the key, of type T). The value of the elements in a multiset cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

Internally, the elements in a multiset are always sorted following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).

Multiset containers are generally slower than unordered_multiset containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.

Multisets are typically implemented as binary search trees.

Associative Containers – multiset

```
  std::multiset<int> mymultiset;
  std::multiset<int>::iterator it;

  // set some initial values:
  for (int i=1; i<=5; i++) mymultiset.insert(i*10);  // 10 20 30 40 50

  it=mymultiset.insert(25);

  it=mymultiset.insert (it,27);    // max efficiency inserting
  it=mymultiset.insert (it,29);    // max efficiency inserting
  it=mymultiset.insert (it,24);    // no max efficiency inserting (24<29)

  int myints[]= {5,10,15};
  mymultiset.insert (myints,myints+3);

  std::cout << "mymultiset contains:";
  for (it=mymultiset.begin(); it!=mymultiset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

//myset contains: 5 10 10 15 20 24 25 27 29 30 40 50
```

Associative Containers – map

Maps are associative containers that store elements formed by a combination of a **key value and a mapped value**, following a specific order.

In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key. The types of key and mapped value may differ, and are grouped together in member type value_type, which is a pair type combining both:

typedef pair<const Key, T> value_type;

Associative Containers – map

Internally, the elements in a map are always sorted by its key following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).

map containers are generally slower than unordered_map containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the bracket operator ((operator[]).

Maps are typically implemented as binary search trees.

Associative Containers – map

```
std::map<char,int> mymap;

// first insert function version (single parameter):
mymap.insert ( std::pair<char,int>('a',100) );
mymap.insert ( std::pair<char,int>('z',200) );

std::pair<std::map<char,int>::iterator,bool> ret;
ret = mymap.insert ( std::pair<char,int>('z',500) );
if (ret.second==false) {
  std::cout << "element 'z' already existed";
  std::cout << " with a value of " << ret.first->second << '\n';
}

// second insert function version (with hint position):
std::map<char,int>::iterator it = mymap.begin();
mymap.insert (it, std::pair<char,int>('b',300));  // max efficiency inserting
mymap.insert (it, std::pair<char,int>('c',400));  // no max efficiency inserting

// third insert function version (range insertion):
std::map<char,int> anothermap;
anothermap.insert(mymap.begin(),mymap.find('c'));

// showing contents:
std::cout << "mymap contains:\n";
for (it=mymap.begin(); it!=mymap.end(); ++it)
  std::cout << it->first << " => " << it->second << '\n';

std::cout << "anothermap contains:\n";
for (it=anothermap.begin(); it!=anothermap.end(); ++it)
  std::cout << it->first << " => " << it->second << '\n';
```

Associative Containers – multimap

Multimaps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order, and **where multiple elements can have equivalent keys**.

In a multimap, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key. The types of key and mapped value may differ, and are grouped together in member type value_type, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a multimap are always sorted by its key following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).
multimap containers are generally slower than unordered_multimap containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.
Multimaps are typically implemented as binary search trees.

Links to sources used in this presentation:

http://www.cplusplus.com/reference/stl/
http://www.cplusplus.com/reference/array/array/
http://www.cplusplus.com/reference/vector/vector/
http://stackoverflow.com/questions/17299951/c-vector-what-happens-whenever-it-expands-reallocate-on-stack
http://www.cplusplus.com/reference/deque/deque/
http://www.cplusplus.com/reference/forward_list/forward_list/
https://en.wikipedia.org/wiki/Sequence_container_(C%2B%2B)
http://www.cplusplus.com/reference/list/list/
http://www.cplusplus.com/reference/set/set/
http://www.cplusplus.com/reference/map/map/
http://www.cplusplus.com/reference/map/multimap/