Short Course on Programming in C/C++

Organized by Onur Pekcan

Contributor Selim Temizer   Instructor Hasan Yılmaz

MİTENG
MITAS ENGINEERING INC.
Eski Güvercinlik Yolu No : 113 Gazi 06560 Ankara TURKIYE
P: + 90 312 296 2000 :: f: + 90 312 296 2999
www.miteng.com

MOBILITY

Week 3 – Lecture2

Today

We will cover;

Standard Template Library (cont.)

- **Associative Containers**
  - set, multi-set
  - map, multi-map
- **Operations/Utilities**
  - Iterator
  - Algorithm

Associative Containers

- set, multi-set
- map, multi-map

set

- Unordered sets are containers that store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.

➢ (constructor)	Construct unordered_set (public member function)
➢ (destructor)	Destroy unordered set (public member function)
➢ operator=	Assign content (public member function)
➢ empty	Test whether container is empty (public member function)
➢ size	Return container size (public member function)
➢ max_size	Return maximum size (public member function)
➢ begin	Return iterator to beginning (public member type)
➢ end	Return iterator to end (public member type)
➢ cbegin	Return const_iterator to beginning(public member function)
➢ cend	Return const_iterator to end (public member function)

multi-set

- Unordered multisets are containers that store elements in no particular order, allowing fast retrieval of individual elements based on their value, much like unordered_set containers, but allowing different elements to have equivalent values.

➢ **(constructor)**	Construct unordered_multiset (public member function)
➢ **(destructor)**	Destroy unordered multiset (public member function)
➢ **operator=**	Assign content (public member function)
➢ **empty**	Test whether container is empty (public member function)
➢ **size**	Return container size (public member function)
➢ **max_size**	Return maximum size (public member function)
➢ **begin**	Return iterator to beginning (public member type)
➢ **end**	Return iterator to end (public member type)
➢ **cbegin**	Return const_iterator to beginning (public member type)
➢ **cend**	Return const_iterator to end (public member type)

map

- Unordered maps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, and which allows for fast retrieval of individual elements based on their keys.
  - **(constructor)**Construct unordered_map (public member function)
  - **(destructor)**Destroy unordered map (public member function)
  - **operator=**Assign content (public member function)
  - **empty**Test whether container is empty (public member function)
  - **size**Return container size (public member function)
  - **max_size**Return maximum size (public member function)
  - **begin**Return iterator to beginning (public member function)
  - **end**Return iterator to end (public member function)
  - **cbegin**Return const_iterator to beginning (public member function)
  - **cend**Return const_iterator to end (public member function)

multi-map

- Unordered multimaps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, much like unordered_map containers, but allowing different elements to have equivalent keys.
  - **(constructor)**Construct unordered_multimap (public member function)
  - **(destructor)**Destroy unordered multimap (public member function)
  - **operator=**Assign content (public member function)
    emptyTest whether container is empty (public member function)
  - **size**Return container size (public member function)
  - **max_size**Return maximum size (public member function)
    beginReturn iterator to beginning (public member type)
  - **end**Return iterator to end (public member type)
  - **cbegin**Return const_iterator to beginning (public member function)
  - **cend**Return const_iterator to end (public member function)

Operations/Utilities

- Iterator
- Algorithm

iterator

Iterator definitions

- In C++, an iterator is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (at least, the increment (++) and dereference (*) operators).
- The most obvious form of iterator is a *pointer*: A pointer can point to elements in an array, and can iterate through them using the increment operator (++). But other forms of iterators exist. For example, each container type (such as a vector) has a specific *iterator* type designed to iterate through its elements in an efficient way.

iterator

- Notice that while a pointer is a form of iterator, not all iterators have the same functionality a pointer has; To distinguish between the requirements an iterator shall have for a specific algorithm, five different *iterator categories* exist:

$$\begin{tikzcd}
\text{RandomAccess} \arrow[r] & \text{Bidirectional} \arrow[r] & \text{Forward} \arrow[r] \arrow[dr] & \text{Input} \\
 & & & \text{Output}
\end{tikzcd}$$

- In this graph, each iterator category implements the functionalities of all categories to its right:

iterator

- Input and output iterators are the most limited types of iterators, specialized in performing only sequential input or output operations.

- Forward iterators have all the functionality of input and output iterators, although they are limited to one direction in which to iterate through a range.

- Bidirectional iterators can be iterated through in both directions. All standard containers support at least bidirectional iterators types.

- Random access iterators implement all the functionalities of bidirectional iterators, plus, they have the ability to access ranges non-sequentially: offsets can be directly applied to these iterators without iterating through all the elements in between. This provides these iterators with the same functionality as standard pointers (pointers are iterators of this category).

Example 1

```
// advance example
#include <iostream>
#include <iterator>
#include <list>

using namespace std;

int main () {
  list<int> mylist;
  for (int i=0; i<10; i++) mylist.push_back (i*10);

  list<int>::iterator it = mylist.begin();

  advance (it,5);

  cout << "The sixth element in mylist is: " << *it << endl;

  return 0;
}
```

algorithm

- The header <algorithm> defines a collection of functions especially designed to be used on ranges of elements.

- A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the STL containers. Notice though, that algorithms operate through iterators directly on the values, not affecting in any way the structure of any possible container (it never affects the size or storage allocation of the container).

algorithm

➢ for_each	Apply function to range (template function)
➢ Find	Find value in range (function template)
➢ Swap	Exchange values of two objects (function template)
➢ swap_ranges	Exchange values of two ranges (function template)
➢ Replace	Replace value in range (function template)
➢ Remove	Remove value from range (function template)
➢ Sort	Sort elements in range (function template)
➢ Merge	Merge sorted ranges (function template)
➢ Min	Return the lesser of two arguments (function template)
➢ Max	Return the greater of two arguments(function template)
➢ min_element	Return smallest element in range (function template)
➢ max_element	Return largest element in range (function template)

Example 2

```
// for_each example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void myfunction (int i) {
  cout << " " << i;
}

struct myclass {
  void operator() (int i) {cout << " " << i;}
} myobject;

int main () {
  vector<int> myvector;
  myvector.push_back(10);    myvector.push_back(20);     myvector.push_back(30);       //push 10,20,30

  cout << "myvector contains:";
  for_each (myvector.begin(), myvector.end(), myfunction);

  // or:
  cout << "\nmyvector contains:";
  for_each (myvector.begin(), myvector.end(), myobject);

  cout << endl;

  return 0;
}
```