Advanced Programming Concepts with C++
CSI2372 – Fall 2017

Jochen Lang
EECS, University of Ottawa
Canada

uOttawa

L'Université canadienne
Canada's university

This Lecture

No reinventing the wheel

- **STL**
  - Sequential container adapters, Ch. 9.6
    - `queue, priority_queue, stack`
  - Associative containers, Ch. 11
    - `map, set, multimap, multiset`
    - C++11 `unordered_map, unordered_set, unordered_multimap, unordered_multiset`
  - Generic algorithms, Ch. 10

Sequential Container Adapters

- **Idea**
  - Have a container behave like something else
- **Two headers**
  - Both `queue` and `priority_queue` are defined in: `<queue>`
  - Stack is defined in `<stack>`
- **Construction**
  - Default constructor creates an empty default container
  - Both queue and stack adapt `deque` by default
  - `priority_queue` adapts a vector by default

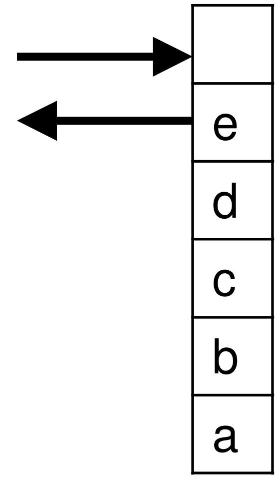
Adapting a Different Container

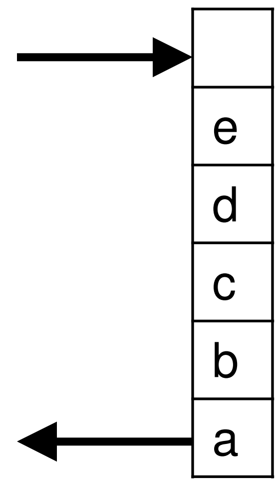
- In the construction a different container may be specified to be adapted
- Template parameter needs to be specified

```
#include <queue>
#include <stack>
using std::vector;
using std::priority_queue;
using std::stack;
// Construction by a vector of 100 elements all = 1
vector<int> iVec( 100, 1);
// Copy the vector as a basis for priority_queue - default
priority_queue<int> pq( iVec.begin(), iVec.end() );
// Copy the vector as a basis for stack - non-default
stack<int, vector<int> > iStack( iVec );
```

Some Methods

- **Stack operation**
  - Placing an element onto the stack `push(element)`
  - Returning the top element `top()`
  - Removing but not returning the top element `pop()`
- **Queue operation**
  - Placing an element into the queue `push(element)`
  - Returning the front element `front()`
  - Returning the back element `back()`
  - Removing but not returning the front element `pop()`





Associative Containers

- **Idea**
  - Store elements in the container based on a key
- **Need to pair up a key and a value**
  - The header `<utility>` defines a type `pair`
  - Pair holds public data members `first` and `second`
  - Default construction or with two initializers

key	**value**
`first`	`second`

Some Operations on Pairs

- **Constructors**
  - Default construction `pair<T1,T2> p`
  - Construction with initialization

    `pair<T1,T2> p( v1, v2 )`
  - Making a new pair with inferring types from arguments `make_pair( v1, v2 )`
- **Comparison**
  - LessThan Comparable: `p1 < p2` defined as `p1.first < p2.first || (!(p1.first < p2.first) && !(p1.first > p2.first) && p1.second < p2.second)`
  - Equality Comparable: `p1 == p2` is true if `p1.first == p2.first && p1.second == p2.second`

Map Type

- **Sorted collection of key-value pairs with unique keys**
  - Commonly red-black trees are used for implementation
- **Construction**
  - Default construction `map<K,V> KVmap`
  - Copy construction `map<K,V> KVmap( oMap )`
  - Copy a range of pairs

    `map<K,V> KVmap( iterA, iterB )`
- **Types in map**
  - Key `map<K,V>::key_type`
  - Value `map<K,V>::mapped_type`
  - Pair `map<K,V>::value_type`

Constraints on keys

– Strict weak ordering (in mathematical notation)

$$!\,(k1 < k1)$$
$$k1 < k2 < k3 \Rightarrow\ k1 < k3$$
$$!\,(k1 < k2 \,\&\, k2 < k1)$$

– Map container uses `operator<` by default

– Keys in maps are `const`

Map Construction

- **Default constructor**
  - empty map
- **Range constructor**
  - as with sequential containers, takes two iterators
- **Copy constructor**
  - copies each element (as with sequential containers)
- **Initializer list constructor (C++11)**
  - as with sequential containers but here with pairs

```
#include <map>
// map with string as a key and int as value
map<string, int> siMap = {{"Smith, John", 31245},
                          {"Doe, Jane", 245876},
                          {"Scott, Stephen",34411}};
```

Inserting and Removing Elements from a `map`

- **Inserting**
    - Will not insert the pair if key is already in map
  - Single pair `KVmap.insert(p)`
  - With hint where to start the search
    `KVmap.insert(iter, p)`
  - Range of pairs `KVmap.insert(iterA, iterB)`
- **Removing**
  - By key `KVmap.erase( k )`
  - At iterator `KVmap.erase( iter )`
  - Range of pairs `KVmap.erase(iterA, iterB)`

Test if Map contains Key and Map Subscripting

  - Count the occurrences of a key `KVmap.count(k)`
    - Note: always 0 or 1 in map
  - Return an iterator to the pair with the key `KVmap.find(k)`
- **Subscripting**
  - Subscripting access the value `KVmap[k] = v`
  - *If key is not in map subscripting will add it!*

Example: Insertion into a Map

```
#include <map>
// map with string as a key and int as value
map<string, int> siMap{{"Smith, John", 31245},
              {"Doe, Jane",245876},
              {"Scott, Stephen",34411}};
siMap["Sobey, Anna"] = 89554; // Add another entry
siMap["Doe, Jane"] = 2; // Update value for existing key
// duplicate key - no insertion
siMap.insert(make_pair("Doe, Jane", 1));
// insert pairs in other map - types must match
map <string, int> oMap;
oMap.insert( siMap.begin(), siMap.end() );
auto iter = oMap.find("Doe, Jane");
if ( iter != siMap.end() ) {
   cout << iter->first << " val: " << iter->second << endl;
   siMap.erase( iter );
}
```

```
map <string, int>::iterator
```

Iterating over a Map

- Similar to sequential container
- Element is a pair and iterator dereference yields a pair

```
#include <map>
map<string, int> siMap; // map with string as a key
// loop over the elements
for ( auto iter = siMap.cbegin();
        iter != siMap.cend(); ++iter ) {
  cout << "Key: " << iter->first << endl;
  cout << "Value: " << iter->second << endl;
}
for (auto si:siMap) {  // use range loop
    cout << "Key: " << si.first << endl;
    cout << "Value: " << si.second << endl;
}
```

```
map <string, int>::
     const_iterator
```

Multiple Entries `Multimap`

- **Key can be inserted multiple times**
  - Occurrences of a key `Mmap.count(k)`
    - May return 0 or a positive integer in multimap
  - Iterator to the first pair with the key `Mmap.find(k)`
  - Identical keys are sorted by the order they were inserted
- **Dealing with multiple entries**
  - Multiple entries with the same key are stored in sequence
    - First entry with a key `Mmap.lower_bound(k)`
    - Last entry with a key `Mmap.upper_bound(k)`
    - Get the pair of first and last `Mmap.equal_range(k)`

Sets: `set` and `multiset`

- **Sets are ordered containers like maps but the key and value are the same**
  - Because key are const, elements can only be read accessed
    - `iterator` and `const_iterator` are const
  - Set operations are not part of set
    - generic algorithms are available for mathematical set operations

Unordered Maps (Hash Maps)

- STL map and multi_map are tree maps
  - I.e., by using a balanced binary tree (e.g., AVL tree)
  - Insertion, removal, find all take O(log n) time
- A hash-map may perform better with a good hash function if load factor is not too high
  - A hash-map is in `<unordered_map>` (C++11)
    - `unordered_map` and `unordered_multimap`
  - A hash-set is in `<unordered_set>` (C++11)
    - `unordered_set` and `unordered_multiset`

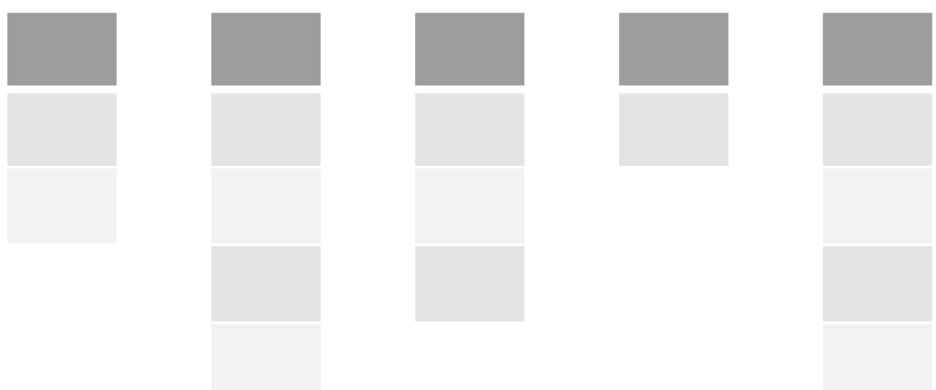
  - Note: Type `hash_map` is not part of the STL but used to be provided by many compilers, e.g., gcc or Visual Studio in a namespace `ext` or `stdext`

Unordered Containers

  - Most operations on unordered containers are the same as for the corresponding associative based on a tree implementation
- **But**
  - No ordering constraints
  - Key is replaced by hash value
    - Unordered container use `==` (instead of < for map)
    - Library supplies hash functions for built-in types and some STL types through `hash<key_type>`
  - Functions to control the set up of the hash table (use of buckets)

Unordered Containers use a Hashtable with Buckets

– Some functions to manage the buckets and to access elements in a bucket
  - `hmap.bucket_count(), hmap.bucket_size(), hmap_bucket(key)`
  - `hmap.load_factor(), hmap.rehash(min_no_buckets), hmap.reserve(num_elements)`



Hashing Keys

– May use own type as key in an unordered container

```
#include <unordered_map>
std::unordered_map<Id,string> hMap;
```

– Need to define both an equality operator for the key class and a hashing function
  - Equality operator

```
struct Id {
std::string d_name;
  int d_id;
  // define the equal operator
  bool operator==(const Id& _oId ) const {
    return (d_id == _oId.d_id && d_name == _oId.d_name);
  }
};
```

Hash Function

– Directly defining a hash function
  • Here relying on the built-in hash for string

```
stuct Id { …
  size_t operator()( const Id& _id ) const {
    return (std::hash<std::string>()(_id.d_name) ^
      std::hash<int>()(_id.d_id)<<1); }  };
```

• **Or**
  – By specializing the template `hash<key_type>`

```
namespace std {
template <> struct hash<Id> {
    size_t operator()( const Id& _id ) const {
    return (std::hash<std::string>()(_id.d_name) ^
        std::hash<int>()(_id.d_id)>>1);
    }
}; } // end of namespace
```

Generic Algorithms

- Algorithms are not part of any container class
  - Unlike Java
- Algorithms also work on other types
  - Types must conform to the STL convention
  - Many work with built-in arrays
- Algorithms work solely with iterators
  - no insertion or removal into a container by the algorithm but change of value or change of element position
  - indirect insertion through an insertion iterator (inserter)

Modifying Elements

- **Example: `fill_n`**
  - fills n elements of a container with a value
    - Remember std algorithms do not create new elements
      - elements must already exist

```
const vector<string>::size_type sz = 10;
vector<string> sVec( sz, "abc" );

std::fill_n( sVec.begin() + 3, 4, string("xyz“));
```

Copying Elements

- **Example: `copy`**
  - copies elements from a source container to a sequential destination container
  - Remember std algorithms do not create new elements
    - elements must already exist

```
std::vector<std::pair<string,int> > siVec;
map<string, int> siMap{{"Smith, John", 31245},
              {"Doe, Jane",245876},
              {"Scott, Stephen",34411}};
siVec.resize(siMap.size());
std::copy( siMap.begin(), siMap.end(), siVec.begin());
for ( const auto& si:siVec ) {
    cout << "Key: " << si.first << endl;
    cout << "Value: " << si.second << endl;
}
```

Deleting Elements

- **Example: `unique`**
  - sorts a container such that there are no consecutive duplicates at the beginning of the container
  - returns an iterator to where no duplicates regions ends
    - Remember std algorithms do not delete elements
      - elements must be deleted separately

```
const vector<string>::size_type sz = 10;
vector<string> sVec( sz, "abc" );
std::fill_n( sVec.begin() + 3, 4, string("xyz"));

vector<string>::iterator lastU =
                      std::unique( iVec.begin(), iVec.end());
```

Sorting

- **Some generic sorting algorithms**
  - Range between random access iterators
    - `sort( iterA, iterB )`
    - `stable_sort( iterA, iterB )`
  - Additionally if range satisfies heap property
    - `sort_heap( iterA, iterB )`

Algorithms Related to Sorting

- **Turn a range between random access iterators into a heap**

```
make_heap( iterA, iterB)
```

- **Merge two sorted ranges between input iterator**

```
merge( iterA1, iterB1,
       iterA2, iterB2, outIter )
```

- **Partition a range between forward iterators**

```
partition( iterA, iterB, predicate )
```

Searching

- **Minimum and maximum in range of forward iterators**
  - `min_element( iterA, iterB )`
  - `max_element( iterA, iterB )`
- **The nth element in range of random access iterators**
  - `nth_element( iterA, iterN, iterB )`
- **Find an element equals a value in range of input iterators**
  - `find( iterA, iterB, val )`
  - With binary search:

    `binary_search(iterA, iterB, val )`

Next

No more Memory leaks

- **Smart pointers and data management**
  - Textbook (Lippman): Chapters 12, 12.1-12.2
  - Smart pointers
  - C++11 smart pointer library types
  - Move constructor and move assignments