C++ Standard Template Library

Alberto Moreno Vega
Àlex Vidal Obiols

EDA 2017 – Spring Semester

Table of contents

Introduction

- C++ library of containers, algorithms and iterators
  - Containers: Manage or contain collections of objects
    - Ej: Vector, map, stack…
  - Algorithms: They act on containers, providing ways of manipulating containers
    - Ej: Sort, search, initializations…
  - Iterators: Used to traverse elements in collections of objects (containers or subsets of containers)

Pair

- Contains a pair of values
- Equivalent to a struct

```
struct pair {
        T1 first;
        T2 second;
};
```

- Used in many other STL containers
- Comparison operators
  - ==, <=, <, >=, >

Pair

- Inclusion
  - #include <utility>
- Initialization
  - std::pair<T1, T2> name (element 1, element 2);
  - Ej: pair<int, char> value (34, ‘R’);
- Access it like a struct
  - int y = value.first;
  - char c = value.second;
- Assignment (with C++ 11 semantics)
  - pair<int,char> value;
  - value = {34, ‘R’};

Iterators

- Used to traverse elements in some collections of objects

- The constructor depends on the collection
  - vector<int>::iterator it; //Iterator for a vector of int
  - vector<string>::iterator it; //Iterator for a vector of string
  - map<int, char>::iterator it; //Iterator for a map of int, char

Iterators

- Useful expressions
  - ++it; //Iterator to the next element
  - --it; //Iterator to the previous element
  - It1 == it2; // Equality comparison
  - It1 != it2; //Inequality comparison
  - *it   //Dereferenced as an rvalue

Iterators

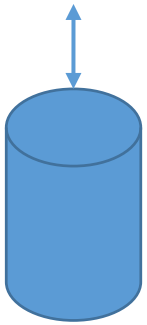
- Example

```
int main() {
   vector<int> v;
   int x;
   while (cin >> x) v.push_back(x);
   for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
      cout << *it << ' ';
   }
   cout << endl;
}
```

Input: 1 2 3 4 5
Output: 1 2 3 4 5

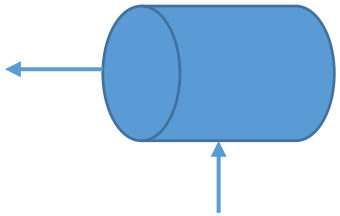
Containers

Stack

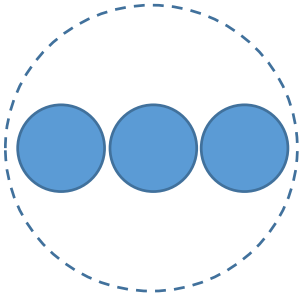


Queue

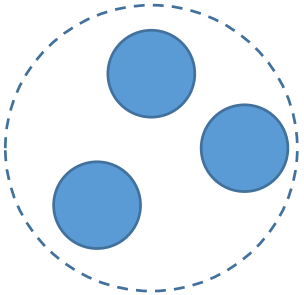
Priority Queue



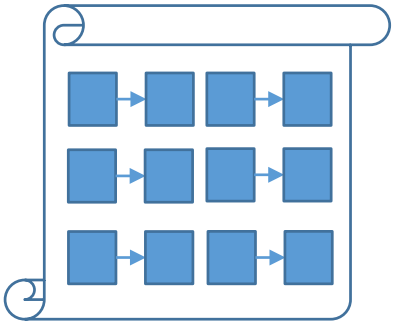
Set



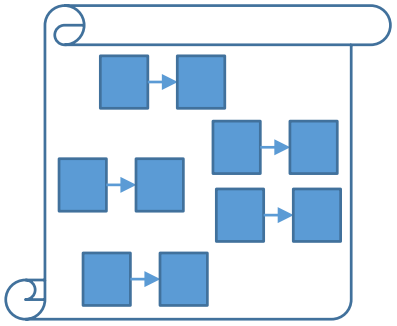
Unordered Set (C++11)



Map



Unordered Map (C++11)

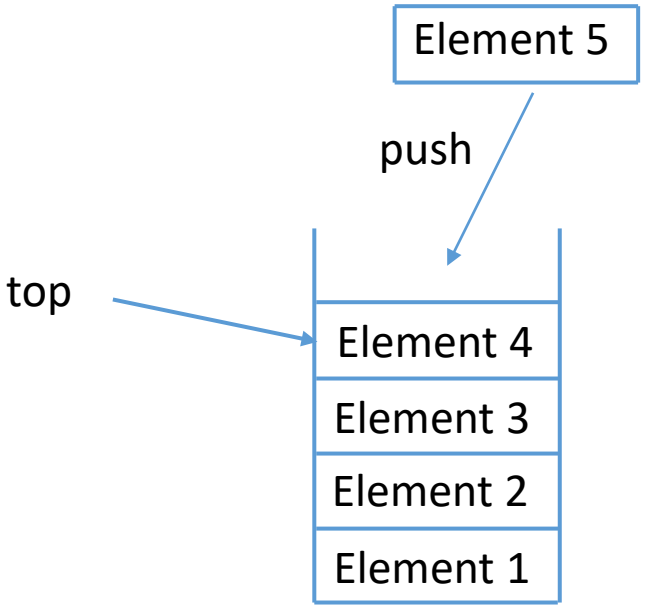


Containers

- The STL implements the most used containers
- The interface to access them is very uniform
- The following methods are defined for all the containers presented here:
  - bool empty() const; $\mathcal{O}(1)$
    - If true, the container has 0 elements.
  - unsigned int size() const; $\mathcal{O}(1)$
    - Returns the amount of elements in the container

Stack

- Container storing the elements in a stack (First In Last Out)



Stack

- Inclusion
  - #include <stack>
- Useful methods:
  - void push(const T& x); $\mathcal{O}(1)$
    - Adds an element to the stack
  - void pop(); $\mathcal{O}(1)$
    - Removes the top element in the stack
  - T& top(); $\mathcal{O}(1)$
    - Returns a **modifiable** reference to the top element. It does NOT remove the element.
  - const T& top() const; $\mathcal{O}(1)$
    - Returns a **constant** reference to the top element. As before, the top element is not removed.

Stack

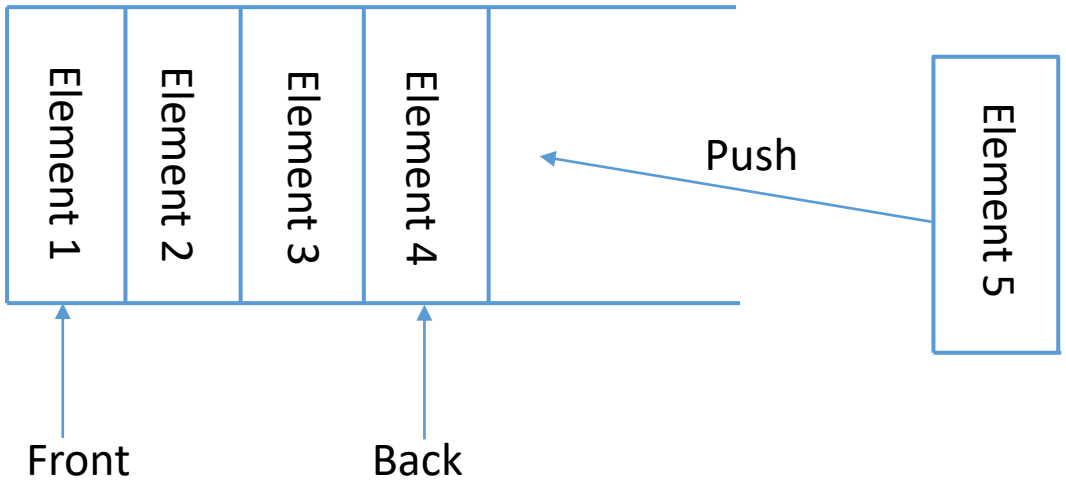
- Example

```
int main() {
   stack<int> s;
   int x;
   while (cin >> x) s.push(x);
   while (not s.empty()) {
      cout << s.top() << endl;
      s.pop();
   }
}
```

Input: 1 2 3 4 5
Output: 5 4 3 2 1

Queue

- Container storing the elements in a FIFO queue (First In First Out)



Queue

- Inclusion
  - #include <queue>
- Useful methods
  - void push(const T& x); $\mathcal{O}(1)$
    - Adds an element to the queue
  - void pop(); $\mathcal{O}(1)$
    - Removes the oldest element in the queue
  - T& front(); $\mathcal{O}(1)$
  - const T& front() const; $\mathcal{O}(1)$
    - Returns a reference to the oldest element in the queue
  - T& back(); $\mathcal{O}(1)$
  - const T& back() const; $\mathcal{O}(1)$
    - Returns a reference to the newest element in the queue

Queue

- Example

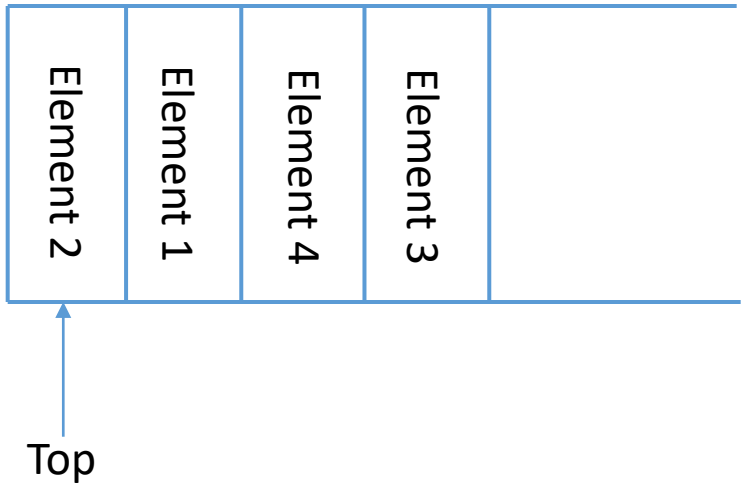
```
int main() {
    queue<int> q;
    int x;
    while (cin >> x) q.push(x);
    while (not q.empty()) {
        cout << q.front() << endl;
        q.pop();
    }
}
```

Input: 1 2 3 4 5
Output: 1 2 3 4 5

Priority Queue

- Container storing elements ordered by their priority. Elements with more priority are accessed first
- By default, if x > y, x has more priority than y

Element 2 > Element 1 > Element 4 > Element 3



Priority Queue

- Inclusion
  - #include <queue>
- Useful methods
  - void push(const T& x); $\mathcal{O}(\log n)$
    - Adds x to the priority queue
  - Void pop(); $\mathcal{O}(\log n)$
    - Removes the element with greatest priority
  - T& top(); $\mathcal{O}(1)$
  - const T& top() const; $\mathcal{O}(1)$
    - Returns the element with greatest priority

Priority Queue

- Example

```
int main() {
    priority_queue<int> pq;
    int x;
    while (cin >> x) pq.push(x);
    while (not pq.empty()) {
        cout << pq.top() << endl;
        pq.pop();
    }
}
```

Input: 7 9 5 3 8
Output: 9 8 7 5 3

Priority Queue

- When constructing a priority queue, it is possible to define how to prioritize elements. By default, priority queues use the operator **less** (less<T>)

- Example for small elements with more priority:
  - priority_queue<int,vector<int>,**greater<int>** > pq;

Set

- A container representing a set of elements
- Elements are not repeated
- Can be traversed by iterators from smaller to greater

```
set<int> s;
…
For (set<int>::iterator it = s.begin(); it != s.end(); ++it)  //prints the elements in s from
        cout << *it << endl;                                  //smaller to greater
```

Set

- Inclusion
  - #include <set>
- Useful functions
  - pair<iterator,bool> insert(const T& x); $\mathcal{O}(\log n)$
    - Inserts an element into the set and returns a pair<iterator,bool>.
      - If the element already existed, the boolean is false and the iterator points to the element in the set.
      - If the element did not exist, the boolean is true and the iterator points to the new element.
    - Ej:

```
set<int> s;
….
pair<set<int>::iterator, bool> p = s.insert(x); // C++11 -> auto p = s.insert(x);
if (not p.second) cout<<“The element “<<*(p.first)<<“ was already in the set”;
else cout << “The element “<<*(p.first)<< “ was not in the set”;
```

Set

- Useful functions
  - iterator begin(); $\mathcal{O}(1)$
    - Returns an iterator to the smallest element
  - iterator end(); $\mathcal{O}(1)$
    - Returns an iterator to the **next** element to the biggest element. Note that dereferencing and accessing this iterator will cause segmentation fault
  - Iterator find(const T& x) const; $\mathcal{O}(\log n)$
    - Searches an element in the set and returns an iterator. If the element exists, the iterator points to the element. Else, the iterator points to end()

Set

- Useful functions
  - void erase(iterator it); amortized $\mathcal{O}(1)$
    - Removes the element pointed by it
  - int erase(const T& x); $\mathcal{O}(\log n)$
    - Removes an element from the set and returns 1 if the element existed, else returns 0

Set

- Example

```
int main() {
   set<int> s1, s2;
   int x;
   while (cin>>x and x!=0) s1.insert(x);
   while (cin>>x and x!=0) s2.insert(x);
   for(set<int>::iterator it = s1.begin(); it != s1.end(); ++it)
      if (s2.find(*it) != s2.end())
         cout << *it << " ";
   cout << endl;
}
```

Input: 1 2 3 4 5 0 3 5 7 9 0
Output: 3 5

Map

- A map is a dictionary of keys *K* and values *V*. Keys are unique (cannot be repeated). The elements are sorted from smaller to greater by key
- Intuitively, keys are used to index values

Key 3 < Key 5 < Key 1 < Key 2 < Key 4

Key 3	Value 3
Key 5	Value 5
Key 1	Value 1
Key 2	Value 2
Key 4	Value 4

Map

- Inclusion
  - #include <map>
- Can be traversed by iterators
  - Ex. map<int,char>::iterator it;
  - (*it).first or it->first accesses the key
  - (*it).second or it->second accesses the value
- Useful functions
  - iterator begin(); $\mathcal{O}(1)$
    - Returns the iterator to the pair with the smallest key
  - iterator end(); $\mathcal{O}(1)$
    - Returns the iterator to the **next** pair with the greatest key

Map

- Useful functions
  - pair<iterator,bool> insert(const pair<K,V>& p); $\mathcal{O}(\log n)$
    - Inserts the pair p into the map and returns a pair of iterator and bool. Similarly to the set, the boolean is false if the key already existed and true otherwise. The iterator points to the element in the map. If the element already existed, the value is **NOT** updated
  - iterator find(const K& k) const; $\mathcal{O}(\log n)$
    - Searches the pair with key k. If it exists, the iterator points to it, else it points to end()
  - void erase(iterator it); amortized $\mathcal{O}(1)$
    - Removes the pair pointed by it
  - int erase(const K& k); $\mathcal{O}(\log n)$
    - If a pair with key k exists, it is removed, returning a 1. Else it returns 0

Map

- It is possible to use the operators ‘[]’ as in the vector class by specifying the key inside the brackets

```
map<char,int> m;
m[‘d’] = 40;
m[‘r’] = m[‘d’] + 6;
```

- If the key did not exist, a new pair is inserted. Else, a reference to the value is returned

Map

- Example

```
int main() {
   map<char,int> m;
   m.insert(pair<char,int>('a', 10));
   m.insert(make_pair('c',30));
   m['d'] = 40;
   map<char,int>::iterator it = m.find('b');
   if (it != m.end())
      m.erase(it);
   m.erase('c');
   for(it = m.begin(); it != m.end(); ++it)
      cout << it->first << " " << it->second << endl;
}
```

Output:

```
a 10
d 40
```

C++ 11

- A major revision for C++
- We will only show a few of the new semantics
- In g++ it is necessary to add the flag –std=c++11

C++ 11

- The compiler can infer the type of a variable with the use of “auto”
  - auto it = set.insert(y);
- Iteration over object collections is now easier

```
vector<int> v;
…
for (int y : v) cout << y << endl; //Prints all the elements of v
for (int& y : v) y *= 2; // Notice the &. Double the value of all the elements in v
for (auto& y : v) cout << y << endl; // Prints all the values of v but now we don’t
                                     specify the type for v
```

- Initialization lists can be used in STL containers
  - vector<int> v = {1, 3, 5, 7};

C++ 11

- Example

```
int main() {
   vector<int> v;
   int x;
   while (cin >> x) v.push_back(x);

   for(int y : v) cout << y << endl; //Prints all the elements in v

   for(int& y : v) y *= 2; // Doubles the value of all the elements in v

   set<set<int>> S = {{2,3}, {5,1,5}, {}, {3}};
   for(auto s1 : S) { //Traverses all the sets in S
      cout << "{ ";
      for(auto s2 : s1)  //Traverses all the elements in one of the sets from S
         cout << s2 << " ";
      cout << "}" << endl;
   }
}
```

C++ 11 – Unordered Set

- A container similar to set, but the elements are not sorted. Implemented with hash tables
- The functions presented previously for set are available for unordered set
- Insert, find and erase methods are linear in worst case, but constant in average (hence the advantage of unordered sets)

C++ 11 – Unordered Map

- A container similar to map, but the elements are not sorted. Implemented with hash tables
- The functions presented previously for map are available for unordered map
- Insert, find and erase methods are linear in worst case, but constant in average (hence the advantage of unordered maps)