Standard Template Library (STL)

Overview

Yngvi Bjornsson

STL

- Collection of useful
  - abstract data types (ADT)
  - generic algorithms
- Implemented as class / function templates
- STL is
  - not a part of the C++ language itself
  - but a part of the C++ standard!

STL Components

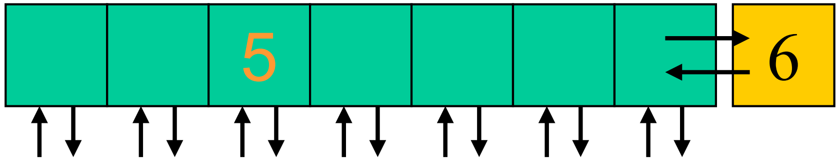
- Six main components
  - Containers
    - Sequence
    - Associative
  - Iterators
    - Constant/Mutable
    - Forward/Bi-directional/Random Access
  - Adaptors
    - Container / Interator / Function
  - Allocators
  - Generic algorithms
  - Function objects

Containers

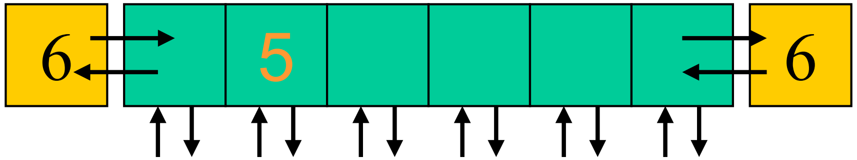
- Sequence Containers
  - Arrange data-items into a linear arrangement.
  - Container types
    - vector<T>
    - deque<T>
    - list<T>
- (Sorted) Associative Containers
  - Fast insert/retrieval of data-items based on keys
  - Container types
    - set<Key>   multiset<Key>
    - map<Key,T>   multimap<Key,T>
  - "multi" containers allow multiple items with same key.

Sequence Containers

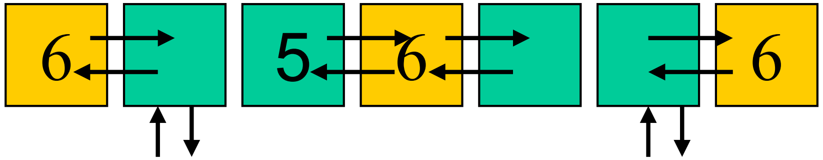
- Vector:
  - Efficient add/delete back, access/modify any element



- Deque:
  - Efficient add/delete front/back, access/modify any element



- List:
  - Efficient add/del. elements anywhere, access/mod. front/back



Common Operations on Seq. Containers

Descr.	**Method**	**V**	**D**	**L**	**Descr.**	**Method**	**V**	**D**	**L**
E front (ref)	**front**	+	+	+	Del all	**clear**	+	+	+
E back (ref)	**back**	+	+	+	Add pos	**insert**	±	±	+
Add E back	**push_back**	+	+	+	Del pos	**erase**	±	±	+
Del E back	**pop_back**	+	+	+		**<**	+	+	+
Add E front	**push_front**	-	+	+		**!=**	+	+	+
Del E front	**pop_front**	-	+	+		**==**	+	+	+
E at subscript	**[ ]**	+	+	-	# E	**size**	+	+	+

```cpp
#include <list>
#include <vector>
#include <deque>
using namespace std;

int main() {
  // STL Vector can use as an array (grows dynamically!)
  vector<int> V1(10), V2;
  for ( int i=0 ; i<10; i++ ){
      V1[i] = i;   // NOTE: cannot use subscript unless element exists!
      V2.push_back(i);
  }
  for ( int i=0 ; i<10; i++ ){ cout << V1[i] << " " << V2[i] << endl; }

  // STL Deque
  deque<int> D1, D2;
  for ( int i=0 ; i<10; i++ ){
      D1[i] = i;  D2.push_front(i);
  }
  for ( int i=0 ; i<10; i++ ){ cout << D1[i] << " " << D2[i] << endl; }
  // STL List (no subscripting)
  list<int> L1, L2;
  for ( int i=0 ; i<10; i++ ){
      L1.push_back(i); L2.push_front(i);
  }
  for ( int i=0 ; i<10; i++ ){
      cout << L1.front() << " " << L2.back() << endl;
      L1.pop_front(); L2.pop_back();
  }
}
```

Iterators

- Standard interface of how to traverse elements in a container.
- Allows us to write generic algorithms that work for any container! (… almost)
- Iterators behave in a way like pointers
  - * (dereference), ++, --, ==, !=, +=, -=
  - Can think of as generalized pointers

Iterators and Containers

- The container classes have methods
  - begin()	Returns an iterator "pointing" to first element
  - end()	Returns an iterator pointing after last element
  - (also rbegin() and rend() for reverse traversal)

- Example:

```
list<int> L;
list<int>::iterator il;
for ( il=L.begin() ; il != L.end() ; ++il ) {
  cout << *il << endl;
}
```

Iterator Kind

- Kind
  - Forward iterators
    - ==, !=, ++, * (dereference)
  - Bi-directional iterators
    - -- (additially to forward iterators)
  - Random access iterators
    - [], +, -, +=, -=   (additionally to bi-directional iterators)
- Containers
  - List (bi-directional), vector and deque (random-access)

Iterator Declaration

- Mutable (default):
  - list<int>::iterator
- Constant
  - List<int>::const_iterator
- Reverse (also mutable)
  - list<int>::reverse_iterator
- Constant and Reverse
  - list<int>::const_reverse_iterator

Container Adaptors

- Container classes implemented on top of the existing containers (adapt their interface)
  - stack (default container: deque)
  - queue (default container: deque)
  - priority_queue (default container: vector)
- Can change underlying container class
  - stack<int, **vector<int>** >
  - Note, need space in between >

Why?

Should not be needed in C++11

Operations on Adapter Containers

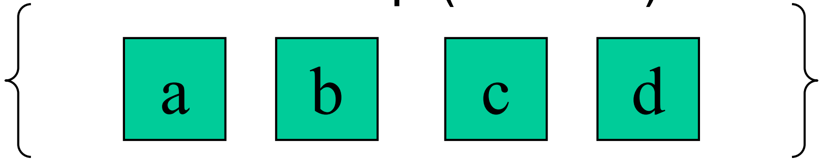
Descr.	Method	S	Q	P
# of elements (E)	**size**	+	+	+
is empty?	**empty**	+	+	+
E at top (ref)	**top**	+	-	+
Add E top / back	**push**	+	+	+
Del E top / front	**pop**	+	+	+
E at front (ref)	**front**	-	+	-
E at back (ref)	**back**	-	+	-

```cpp
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

int main()
{ // STL Stack
  stack<int, vector<int> > S;// Changing default container (note '> >', not  '>>')
  for ( int i=0 ; i<10; ++i ) { S.push(i); }
  for ( int i=0 ; i<10; ++i ) {
    cout << S.top() << " ";
    S.top() = 2 * S.top();
    cout << S.top() << endl;
    S.pop();
  }
  // STL Queue
  queue<int> Q;
  for ( int i=0 ; i<10; ++i ) { Q.push(i); }
  for ( int i=0 ; i<10; ++i ) {
    cout << Q.front() << endl;
    Q.pop();
  }
  // STL Priority Queue
  priority_queue<int> P;
  for ( int i=0 ; i<10; ++i ) { P.push(i); }
  for ( int i=0 ; i<10; ++i ) {
    cout << P.top() << endl;
    P.pop();
  }
}
```

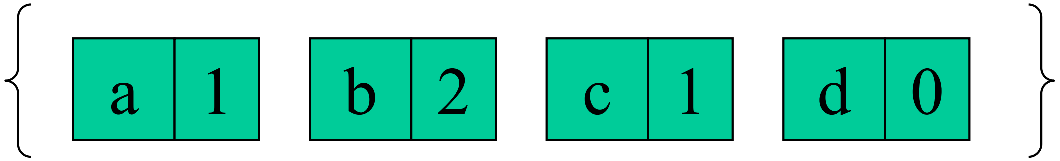
Associative Containers

- set:
  - Add/Delete elements to/from set (no duplications)
  - Can ask of membership (is in set)



{ a b c d }

- map: (associative array)
  - Add/Delete **pairs** to/from map (no duplications of keys)



{ a 1 b 2 c 1 d 0 }

- multiset / multimap essentially the same except duplications of keys allowed.

Common Operations on Associative Containers

Descr.	Method	S	M
# of elements (E)	**size**	+	+
is empty?	**empty**	+	+
Add element	**insert**	+	+
Delete element	**erase**	+	+
Find element (ref)	**find**	+	+
The same	**==**	+	+
Subscript with key	**[]**	+	+

```
#include <iostream>
#include <string>
#include <set>
#include <map>
using namespace std;

int main()
{
  // STL set
    set<string> S;
  S.insert("hi");
  S.insert("how");
  S.insert("are");
  S.insert("you");
  for ( set<string>::iterator is=S.begin() ; is != S.end() ; is++ ) {
    cout << *is << endl;
  }

  //STL map
  map<string,int> M;
  M["Hi"] = 0;
  M["how"] = 1;
  M["are"] = 2;
  M["you"] = 1;
  if ( M.find("how")  !=  M.end() )
    cout << "'how' is in map" << endl;

  for ( map<string,int>::iterator im = M.begin() ; im != M.end() ; ++im ) {
     cout << im->first << " " << im->second << endl; // note use of pair
  }
}
```

Be careful how to to look for elements in a map. Use **find** instead of e.g. something like:

```
if ( M["hello"] == 0 ) {
  …
}
```

Generic Algorithms

- Can operate on a variety of data structures
  - (not limited to STL, e.g. also arrays)

- Four broad main categories
  - Non-mutating sequence algorithms
  - Mutating sequence algorithms
  - Sorting-related algorithms
  - Generalized numeric algorithms

Non-mutating Sequence Algorithms

- Work on a sequence, without changing the elements in the sequence
- Sample algorithms

– find	find an element in range
– count	count number of elem. in range equal to
– for_each	apply a function to each elem in range
– equal	checks if two ranges are the same
– search	search for a sub-sequence in range
– …	… and more …

Find

- Find an element in a range:
  - iterator find(iterator1,iterator2,elem);
- Range (can search part of a container)
  - [iterator1,iterator2)
- Functionality:
  - Looks for first occurrence of ‘elem’ in the range [iterator1, iterator2)
  - Returns an iterator pointing to the element
  - If element not in sequence return iterator2

```
#include <iostream>
#include <string>
#include <list>
#include <algorithm>   // Note, include file
using namespace std;

int main()
{
    list<string> L;

    // Add elements to list
    L.push_back("One");
    L.push_back("Two");
    L.push_back("Three");
    L.push_back("Four");
    L.push_back("Three");
    L.push_back("Two");
    L.push_back("One");

    // Find first "Four", and first "Two" after "Four"
    list<string>::iterator front, back;
    front = find(L.begin(), L.end(), "Four");
    back = find(front, L.end(), "Two");

    // Output "Four Three"
    for ( list<string>::iterator i=front; i != back; i++ ) {
        cout << *i << " ";
    }
    // Output number of "Two" in list.
    cout << endl << count( L.begin(), L.end(), "Two") << endl;
}
```

Mutating Sequence Algorithms

- Work on a sequence, and (possibly) modify it
- Example algorithm:

– copy	copies range from one sequence to another
– fill	sets all element in range to specified value
– random_shuffle	randomly shuffles element in sequence
– remove	removes from range elem. equal to specified value
– replace	replaces elements in range that are equal to a value
– reverse	reverses elements in a range
– unique	eliminates consecutive duplicate elem. in range
– … and many more …	

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;

int main() {
  vector<int> V;

  // Add elements to vector
  V.push_back(1);
  V.push_back(2);
  V.push_back(3);
  V.push_back(4);
  V.push_back(5);
  V.push_back(6);
  V.push_back(7);

  list<int> L(V.size());  // Note, for copy to work, elements in
                          // target range must already exist (overwrites)

  // Copy elements to a list, just for the fun of it
  copy( V.begin(), V.end(), L.begin() );
  // ... and now reverse the list (could also have reversed vector)
  reverse( L.begin(), L.end() );

    // Output the list
  for ( list<int>::iterator i=L.begin(); i != L.end(); ++i ) {
    cout << *i << " ";
  }
}
```

Sorting-Related Algorithms

- Both:
  - sorting a sequence
  - operations applied to sorted containers
- Example
  - sort, partial_sort
  - binary_search
  - merge,
  - set_union, set_intersection, …

```
// Example of sorting-related algorithms
// Output: 1 2 3 4 5
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;

int main() {
    deque<int> Dodd, Deven;

    // Add elements to deques
    Dodd.push_back(1);
    Dodd.push_back(3);
    Dodd.push_back(5);
    Dodd.push_back(3);
    Dodd.push_back(1);
    Deven.push_back(2);
    Deven.push_back(4);
    // Sort and "remove" duplicates (moves to back)
    sort(Dodd.begin(), Dodd.end());
    deque<int>::iterator end = unique(Dodd.begin(), Dodd.end());

    // Merge odd and even numbers
    int sizeD = Deven.size()+(end-Dodd.begin());
    deque<int> D(sizeD);
    merge(Dodd.begin(), end, Deven.begin(), Deven.end(), D.begin() );

    // Output deque
    for ( deque<int>::iterator i=D.begin(); i != D.end(); ++i ) {
       cout << *i << " ";
    }
}
```

Numeric Algorithms

- Four general numeric algorithms
  - accumulate
  - partial_sum
  - adjacent_difference
  - inner_product
- Example:
  - int a[2] = {1,2}, b[2] = {10, 20};
    cout << inner_product(&a[0],&a[2],&b[0],&b[2],0);

Function Objects (Functors)

- A class with the () operator defined
  - Pre-existing or user defined.
- Can pass as a parameter to many of the generic algorithms
  - Modify the computation behaviour
- For example,
  - Sort in a reverse order
    - sort(V.begin(),V.end(),**greater<int>()**);
  - Find using a different binary operator than ==
  - Accumulate using operator different than +
- Some predicates pre-defined, but can supply our own.

Important STL Additions in C++11

- Sequence containers
  - array
  - forward_list
- Unsorted associative containers (hashing based)
  - unordered_set (and unordered_multiset)
  - unordered_map (and unordered_map)
- Also, the auto keyword and for range-based for loops especially useful for iterating all containers.

```cpp
#include <iostream>
#include <array>
#include <list>
#include <unordered_map>
#include <utility>
using namespace std;

int main()
{
    array<int, 3> A;             // example C++11 fixed_size array declaration
    unordered_map<int, int> M; // example C++11 unordered map declaration

    list< pair<int,int> > L;
    L.push_back( make_pair(1,2) );
    L.push_back( make_pair(2,4) );

    // Iterate over container pre C++11.
    for ( list< pair<int,int> >::iterator it = L.begin(); it != L.end(); ++it ) {
       cout << it->first << ' ' << it->second << endl;
    }

    // Iterate over container post C++11.
    for ( auto e : L ) {
       cout << get<0>(e) << ' ' << get<1>(e) << endl;    // can also use first/second as above
    }
}
```

Summary

- Overview of STL
  - Containers
  - Iterators
- Some of the containers have
  - additional methods not covered here
  - e.g. can splice lists
- For a complete reference see e.g.:
  - http://en.cppreference.com