Overview of the Standard Template Library (STL)

What is STL?

- ANSI standard library for commonly occurring C++ classes
- Makes extensive use of the C++ template mechanism
- Lean and mean (no frills attached)
- Originated at Hewlett Packard Research laboratories

2

STL Components (1/2)

- Containers (collections)
  - Placeholders for related groups of objects of a given type
- Algorithms
  - Generic operations to be performed on containers
- Iterators
  - Intermediaries between containers and algorithms

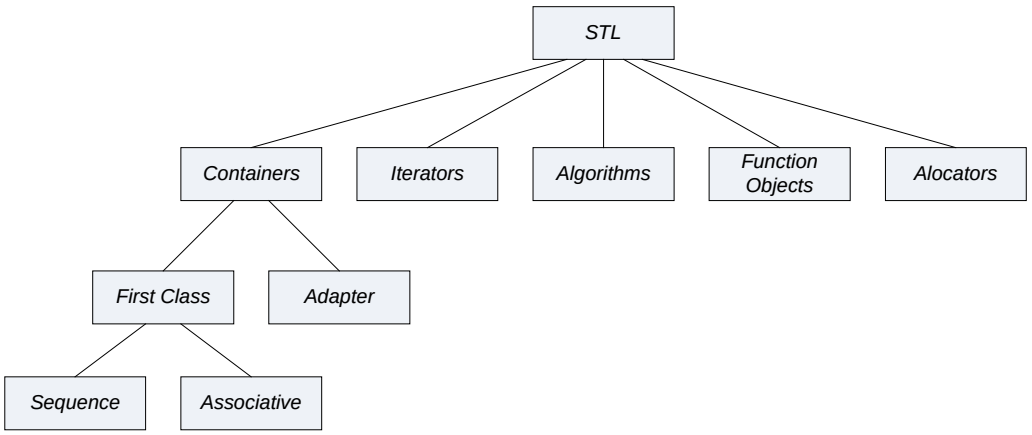
3

STL Components (2/2)

- Function objects
  - Similar to Command design pattern
- Adaptors
  - Change the interface of containers, algorithms and iterators
- Allocators
  - For controlling storage management

4

Overview



5

Containers (1/2)

- All containers organise a collection of objects of the same type
- Sequence containers follow a strictly linear regime
- Sorted associative containers allow fast retrieval of objects based on keys

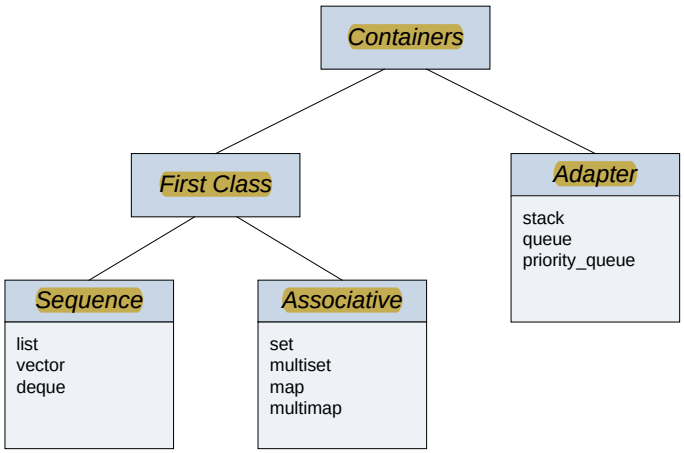
6

Containers (2/2)

- Adaptors are components that change the interface of other components

7

Main Container Types



8

Algorithms (1/2)

- Can group algorithms based on their semantics
- Nonmutating sequence do not change contents of container (read-only)
- Mutating sequence modify container on which they operate (write-enabled)

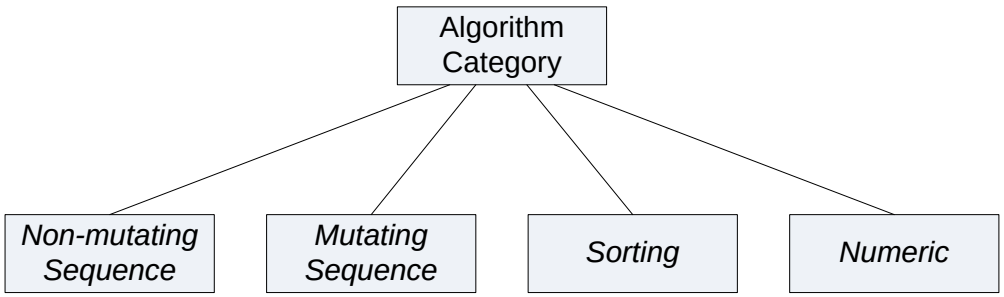
9

Algorithms (2/2)

- Sorting-related (sort, merge, searching) applicable to sorted sequences
- Generalised numeric algorithms (a small collection of simple maths stuff)

10

Algorithm Category



11

Iterators

- Pointer-like objects
- Algorithms use them to traverse containers
- Concept of iterator range is important e.g. [first, end)

12

Iterator Categories (1/2)

- Input iterator
  - Read one element at a time (in forward direction only)
- Output iterator
  - Writes one element at a time (in forward direction only)
- Forward iterator
  - Combines functionality of input and output iterators

All standard containers support at least forward iterator types.

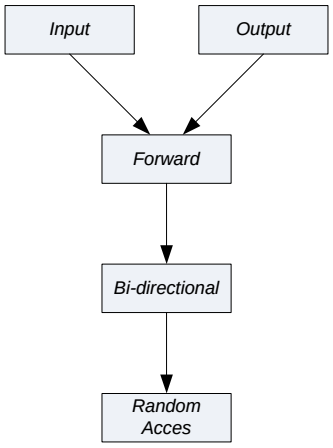
13

Iterator Categories (2/2)

- Bi-directional
  - A forward iterator that can also move backwards
- Random access iterator
  - A 'jumpy' bi-directional iterator

14

Hierarchical Iterator Categories



15

Using Iterators

- Iterator type nested in container class
- Use operator != to compare iterators
  (operator < is not supported)
- Dereference iterator to get the current value

16

General Scheme

```
#include <container>
using namespace std;

// container is an imaginary STL container.
container<int> c;
...

// Iterator type nested in container class.
container<int>::iterator iter;
const container<int>::iterator end = c.end();

// Use operator != in comparisons.
for (iter = c.begin(); iter != end; iter++)
{
  // Dereference to get current value.
  cout << *iter << endl;
}
```

17

Function Objects

- This is an object that encapsulates a function
- Create such an object by overloading the function call operator ‘operator ()’
- Useful (saves code duplication and makes code more reusable)
- Similar to the Command design pattern (Gamma)

18

Adaptors

- Modify the interface of other components
- Applies to containers, iterators and functions
- Possible to produce new classes with restricted interfaces (e.g. stack)
- Useful for customisation purposes

19

Allocators

- Encapsulate information about memory models
- There exists a Default Allocator Interface
- Types: reference, const_reference, pointer and const_pointer
- Default allocator uses standard new and delete
- Can create custom allocators that for example allocates memory in a fixed memory pool (faster but less memory efficient)

20

Some Examples

- Vector and vector iterator
- List and list iterator
- Map
- Algorithms: copy and find

21

Vector and Vector Iterator

- Array of T
- Provides random-access to elements
- Only possible to add elements at the back
- Vector offers random-access iterator
- Include <vector> header file

Just like arrays, vectors use contiguous storage locations for their elements — > Enable random-access

But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

22

Example
Vector

```
#include <vector>
using namespace std;

// Declare vector with 2 elements.
vector<double> v(2);

// Write access to elements.
v[0] = 1;
v[1] = 2;
v.push_back(3); // Increase size of vector!

// Read access to elements.
cout << v[0] << v[1] << v[2] << endl;
```

23

Example
Vector Iterator

```
// Get (random access) iterator.
vector<double>::iterator iter = v.begin();

// Use iterator for random access.
cout << iter[0] << iter[1] << iter[2] << endl;

// Use iterator for traversal.
const vector<double>::iterator end = v.end();
for (iter = v.begin(); iter != end; iter++)
{
  cout << *iter << endl; // Dereference iterator.
}
```

24

List and List Iterator

- Doubly linked list of T
- Possible to add and remove elements at the front, in the middle and at the back of the list
- No random access, iterator jumps from one element to the next
- Include <list> header file

Because they are not stored in contiguous location —> No random access

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.

25

Example
List and List Iterator

```
#include <list>
using namespace std;

list<int> l;

// Add data.
l.push_back(2);
l.push_back(3);
l.push_front(1);

// Use iterator for traversal.
list<int>::iterator iter;
const list<int>::iterator end = l.end();
for (iter = l.begin(); iter != end; iter++)
{
  cout << *iter << endl;
}
```

26

Strengths and Limitations of STL

- Useful in that the programmer does not have to reinvent the wheel
- Low-level reuse only (more sophisticated classes are absent)

27