STL: Containers

Miro Jurišić
meeroh@meeroh.org

Why use STL containers?

- Type-safe
- Feature-rich
- Written and debugged by someone else

Which containers are available?

- Standard sequence: vector, string, deque, list
- Standard associative: set, multiset, map, multimap
- Non-standard sequence: array, slist, rope
- Non-standard associative: hash_set, hash_multiset, hash_map, hash_multimap
- Other standard containers: C arrays, bitset, valarray, stack, queue, priority_queue

Choosing containers

- Do you have to use a standard container?
- Do you need to insert at arbitrary places?
- Do you rely on the order of items?
- Do you need random access?
- Do you need bidirectional access?
- Do you need stable iterators?
- Do you need to interface with C APIs?
- What are your performance requirements?
- What are your thread safety requirements?

Iterators

- Use iterators to refer to items in containers
- Use iterators to walk over items in containers
- unidirectional vs. bidirectional
- forward vs. reverse
- sequential vs. random access
- input vs. output
- ++, –, etc.
- distance()

vector

- storage is guaranteed to be contiguous
- size() vs. capacity(); reserve(); empty(); resize(); clear()
- constructors: default, copy, fill, iterator
- push_back(), pop_back(), at(), []
- insert(), erase()
- begin(), end(), rbegin(), rend()
- all vector iterators are invalidated on any change

```
vector <int>   v1;
v1.push_back (5);

vector <int>   v2 (v1);
v2.push_back (7);
```

```
vector <float> v3 (5, 2.0);
vector <float> v4 (v2.begin(), v2.begin() + 1);

v4.resize (10, 0.1);
v3.clear ();
v2.reserve (20);

v1.pop_back ();
v2.insert (v2.begin (), 10);
v4.erase (v3.begin (), v3.begin () + 3);
```

deque

- constant time insertion at front and back
- empty(); resize(); clear()
- constructors: default, copy, fill, iterator
- push_back(), pop_back(); push_front(), pop_front()
- insert(), erase()
- begin(), end(), rbegin(), rend()
- all deque iterators are invalidated on any change, except for beginning and end removals

```
deque <int>   d1;
d1.push_back (5);
d1.push_front (4);

deque <int>   d2 (d1);
d2.push_back (7);
d2.push_front (8);
```

```
deque <float> d3 (5, 2.0);
deque <float> d4 (d2.begin(), d2.begin() + 1);

d3.clear ();

d1.pop_back ();
d1.pop_front ();
d2.insert (d2.begin (), 10);
d4.erase (d3.begin (), d3.begin () + 1);
```

list

- constant time insertion everywhere
- empty(); resize(); clear()
- constructors: default, copy, fill, iterator
- push_back(), pop_back(); push_front(), pop_front()
- insert(), erase(), splice(), remove(), unique(), merge(), sort(), reverse()
- begin(), end(), rbegin(), rend()
- no list iterators are invalidated, except for those referring to a deleted element

```
list <int>   l1;
l1.push_back (5);
l1.push_front (4);

list <int>   l2 (l1);
l2.push_back (7);
l2.push_front (8);
```

```
list <float> l3 (5, 2.0);
list <float> l4 (l2.begin(), ++l2.begin());

l3.clear ();

l1.pop_back ();
l1.pop_front ();
l2.insert (l2.begin (), 10);
l3.erase (l3.begin (), l3.end ());

l2.sort();
l1.reverse ();
l3.sort ();
l1.splice (++l1.begin (), l2);
l1.sort ();
l1.unique ();
```

string and wstring

- one template: basic_string
- string: basic_string <char>; wstring: basic_string <wchar_t>
- size() vs. capacity(); reserve(); empty(); resize(); clear()
- constructors: default, substring, data buffer, NUL-terminated data buffer, fill, iterator
- push_back(), at(), []
- pop_back() non-standard
- insert(), erase(), append(), +, replace(), find(), rfind(), find_{first,last}_{of,not_of}, substr()
- begin(), end(), rbegin(), rend()
- all string iterators are invalidated on any change
- c_str() is guaranteed to be contiguous (can be passed into C APIs)

```
string   s1;
s1.push_back ('a');

string   s2 (s1);
```

```
s2 = s1 + "b";

string   s3 (10, 'z');
string   s4 (s3, 1, 3);

s2.insert (3, s3);
s3.erase (s3.begin (), s3.begin () + 3);
s3.erase (0, 3);

string   s5 ("ABBA");
s5 [s5.find_first_of ('A')] = 'D';
s5 [s5.find_last_of ('B')] = 'C';
```

set, multiset

- Sorted associative containers (values are also keys)
- size(); empty(); clear()
- constructors: default, copy, iterator; optional comparator
- insert(), erase(), find(), count(), lower_bound(), upper_bound(), equal_range()
- begin(), end(), rbegin(), rend()
- no iterators are invalidated except ones pointing to a removed element

```
set <char>   s1;
s1.insert ('a');

set <char>   s2 (s1);
s2.insert ('b');

multiset <char>   s3 (s2.begin (), s2.end ());
s3.insert ('b');

set <char>::iterator i1 = s2.find ('b');
set <char>::iterator i2 = s1.lower_bound('b');
```

map, multimap

- Sorted associative containers
- size(); empty(); clear()
- constructors: default, copy, iterator; optional comparator
- [], insert(), erase(), find(), count(), lower_bound(), upper_bound(), equal_range()
- begin(), end(), rbegin(), rend()
- no iterators are invalidated except ones pointed to a removed element

```
map <int, string>   m1;
m1 [0] = string ("January");
m1 [1] = string ("February");

map <int, string>   m2 (m1);
m2 [1] = string ("Feb");
m2 [2] = string ("Mar");

string s = m2 [1];
map <int,string>::iterator i1 = m2.find (11);
```

array, slist, rope

- boost::array: similar to vector, but can't be resized
- slist: similar to list, but single-linked
- rope: similar to string, but better suited to very long strings
- efficient for operations on large chunks of the string
- assignment, concatenation, substring performance independent of string length

```
boost::array <int, 4>       a;
a [0] = 1;
a [1] = 2;

vector <int>  v (a.begin (), a.end ());
```

hash_*

- hash table variants of standard associative containers
- in most cases considerably faster than sorted associative containers
- average constant time on most operations (worst case linear)

container adaptors: stack, queue, priority_queue

- container adaptors give new interfaces to existing containers
- stack: LIFO, queue: FIFO
- priority_queue: sorted queue
- Layered on top of a container (typically deque or vector)

bitset

- not a container, just a utility class
- packed fixed size array of bits
- implements bitwise operators