CS2704

Topic 16
Standard Template Library

Outline

• STL Components
• Sequence Containers
• Iterators
• Sorted Associative Containers

STL Components

• Containers - classes for objects that contain
• Generic algorithms - functions that work on
different types of containers
• Iterators - “pointers” into containers
• Function objects
• Adaptors - classes that “adapt” other classes
• Allocators - objects for allocating space

Sequence Containers

• Arrays - random access, fixed length,
constant time access
• vector<T> - random access, varying length,
constant time insert/delete at end
• deque<T> - random access, varying length,
constant time insert/delete at either end
• list<T> - linear time access, varying length,
constant time insert/delete anywhere in list

Vector Example

```cpp
#include <iostream>
#include <vector>
#include <assert>

vector<char> vec(char *s) {
    vector<char> x;
    while(*s != '\0') x.push_back(*s++);
    return x;
}
```

Vector Example (2)

```cpp
int main() {
    vector<char> v1 = vec("01234");
    vector<char>::iterator i = v1.begin();
    while (i != v1.end()) v2.push_back(*i++);
    assert(v1 == v2);
    v1 = vec("01234");
    v2 = vec("\n");
    i = v1.begin();
    while (i != v1.end()) v2.insert(v2.begin(),*i++);
    assert (v2 == vec("45210"));
    }
```
Use of Iterator

- Type defined by container class
  `vector<T>::iterator i;`
- Getting specific iterator values
  - Get first or last: `v.begin()`, `v.end()`
  - Move to next: `i++`, `++i`
- Dereference to get value “pointed” to: `*i`
- Equality and inequality
  - Note: `i == j` and `*i == *j` are different

Vector and Insert

- Like array that can increase size
- Insert at end (`push_back`) most efficient
- Insert elsewhere requires shifting data
- Vector doubles in size if insert after “full”
- Find current size of `v` with `v.capacity()`
- Set size with `v.reserve(n) --- won't shrink`

Vector and Delete

- Remove last element: `v.pop_back()`
- Remove element pointed to by iterator `i`
  with `v.erase()`
  - Requires shifting data
  - Invalidates iterators to positions past iterator
    - `v.erase(i++) // doesn't work`
- Remove range of values with `v.erase(fst, lst)`

Vector Constructors

```cpp
vector<T> vector; // empty vector
vector<T> vector(n, value); // vector with n copies of value
vector<T> vector(n); // vector with n copies of default for T
```

(Some) Vector Methods

```cpp
size_type size() const; // number of elements in vector
bool empty() const; // true if no elements
reference front(); // returns reference to first element
reference back(); // returns reference to last element
reference operator[](size_type n); // nth entry
```

Const Iterators

- Constant iterator used when object is const
- Typical for parameters
- Type is defined by container class
  `vector<T>::const_iterator`
Container Comparison

- Two containers of same type equal if
  - Have same size
  - Elements in corresponding positions are equal
- Type in container must have equality operator
- For other comparisons need operator<

Container Assignment

- All containers have assignment operator
- Also have \texttt{v.assign(fst, lst)} to assign a range to \texttt{v}

Deque Class

- Efficient insert/delete from either end
- Add a \texttt{push_front} method
- Most methods and constructors the same as for vector

List Class

- Essentially a doubly linked list
- Not random access, but constant time insert and delete
- Some differences in methods from vector and deque (ex., no \texttt{operator[]})
- Insertions and deletions do not invalidate iterators

Associative Arrays

- A standard array is indexed by numeric type
  - A[0],…,A[Size]
  - Dense indexing
- An associative array can be indexed by any type
  - A[“alfred”], A[“judy”]
  - Sparse indexing

Sorted Associative Containers

- Values in container sorted by a \texttt{Key} type
- \texttt{set<Key>} - collection of unique \texttt{Key} values
- \texttt{multiset<Key>} - possibly duplicate keys
- \texttt{map<Key, T>} - collection of \texttt{T} values indexed by unique \texttt{Key} values
- \texttt{multimap<Key, T>} - possibly duplicate keys
Orders for Sorting

- STL makes assumptions about orders in sort functions and sorted associative containers
- Ideally, want a strict total ordering:
  - For every x, y, z, if x < y and y < z then x < z
  - For every x and y, then only one of x < y, y < x, and x = y is true.
- Note: cannot be that x < x

Orders for Sorting (2)

- Actually, use a weaker notion of order
- Define relation E from a relation R by
  \( x \ E \ y \) iff both \( x \ R \ y \) and \( y \ R \ x \) are false
- A relation R is a strict weak ordering if it is transitive, asymmetric and E is an equivalence relation

Example Order

```cpp
class Name {
public:
    string last_name;
    string first_name;
};

class LastNameLess {
public:
    bool operator<(const Name& n1, const Name& n2) {
        return n1.last_name < n2.last_name;
    }
};
```

Example Order (2)

- Using LastNameLess,
  - Zephram Alonzo < Alfred Zimbalist
  - Alonzo Church is equivalent to Bob Church
- Notice that equivalence defined this way is not the same as \( \text{operator}== \)

Special Function Objects

- If have \( \text{operator}< \) for a class T then can use special template class to build order function objects
- \( \text{less}<\text{T}> \) assumes T has an \( \text{operator}< \)
- In header file \( \text{function.h} \)

Default Template Arguments

- Can specify a default argument to template
- Default used if a specific class not given
- Ex. For set class:
  ```cpp
  template<\text{class Key, class Compare = \text{less}<\text{Key}>,
  class Alloactor = \text{allocator}>
  ```
- Can say \( \text{set}<\text{Name}, \text{LastNameLess}> \) or \( \text{set}<\text{Name}> \) if \( \text{operator}< \) defined on \( \text{Name} \)
Sets and Multisets

- Both sets and multisets store key values
- Both require order as defined above
- Set only allows distinct objects (by order)
- Multiset allows distinct objects

Set Constructors

```cpp
set(const Compare& comp = Compare());

template<class InputIterator>
set(InputIterator first, InputIterator last,
   const Compare& comp = Compare());

set(const set<Key, Compare, Allocator>& other);
```

Set Example

```cpp
#include <list>
#include <set>
#include <cassert>

// transfer non-null characters to list
list<char> lst(char* s) {  
  list<char> x;
  while (*s != '\0') x.push_back(*s++);
  return x;
}
```

Multiset Example

```cpp
#include <list>
#include <set> // may be <multiset> for g++
#include <cassert>

// transfer non-null characters to list
list<char> lst(char* s) {  
  list<char> x;
  while (*s != '\0') x.push_back(*s++);
  return x;
}
```

Set Example (2)

```cpp
int main() {
  list<char> lst1 = lst("dogs love food");
  // copy list to set
  set<char> set1;
  list<char>::iterator i = lst1.begin();
  while (i != lst1.end()) set1.insert(*i++);
  // copy set to list
  list<char> lst2;
  set<char>::iterator k = set1.begin();
  while (k != set1.end()) lst2.push_back(*k++);
  assert(lst2 == lst("dogs love food"));
}
```

Multiset Example (2)

```cpp
int main() {
  list<char> lst1 = lst("dogs love food");
  // copy list to multiset
  multiset<char> mset1;
  list<char>::iterator i = lst1.begin();
  while (i != lst1.end()) mset1.insert(*i++);
  // copy multiset to list
  list<char> lst2;
  multiset<char>::iterator k = mset1.begin();
  while (k != mset1.end()) lst2.push_back(*k++);
  assert(lst2 == lst("dogs love food"));
}
```
Insert and Erase Methods

- Can insert and erase in two ways
  - By value
    ```cpp
    set1.erase(k); // k is a Key variable
    ```
  - At iterator
    ```cpp
    set1.erase(i); // i is an iterator
    ```

Accessor Methods

- `find(Key)` - returns iterator to an element with given value, equals `end()` if not found
- `lower_bound(k)` - returns iterator to first position where k could be inserted and maintain sorted order
- `upper_bound(k)` - iterator is to last such position

Maps and Multimaps

- Associative arrays on given Key type
- Map requires unique Keys (by def of order)
- Multimap allows duplicate Keys
- Map is like set that holds key-value pairs, which are only ordered on the keys
- Additional operator: `map1[k] = v`

Values in Maps

- `map<Key, T>` holds `pair<const Key, T>`
- Once pair inserted can only change T value
- Pair class has public member fields `first`, `second`
- To create object in map use pair constructor
  `pair<const string, string>("333-33-3333", "Jim")`
- Can also use function `make_pair`

Inserting in Maps and Multimaps

- Insert value (also insert using iterator “hint”)
  ```cpp
  map< string, string > mp1;
  mp1::insert(pair< const string, string >("222-22-2222", "Jenny"));
  ```
- Multimap allows duplicate keys
  ```cpp
  multimap< string, string > mp1;
  mp1::insert(pair< const string, string >("blue", "Jenny");
  ```

Finding Data in Map

- Use `find(Key)` function to find entry by key
  ```cpp
  map< string, string > mp;
  ... // insert some values
  map< string, string >::iterator m_i;
  m_i = mp.find("222-22-2222");
  if ( m_i != mp.end() ) // do something with entry
  ```
- Can manipulate entry
  ```cpp
  (*m_i).first // get key value, cannot be changed (const)
  (*m_i).second // data value, may be changed
  ```
Finding Data in Multimap

- find method only guaranteed to find a value with key
- lower_bound method finds first with key
- upper_bound finds last value with given key
- Use iterator to look at each of duplicate values

Subscripting in Maps

- Map allows use of subscript mp[k] = t
  - If no pair with key k, then pair (k,t) inserted
  - If pair (k,0) exists, replace 0 with t
- If no pair with key k exists in mp the expression mp[k] will insert a pair (k,T())
- Ensures that mp[k] always defined
- Subscripting not defined for multimaps

Iterators

- Several kinds of iterators
- Correspond to assumptions made by generic algorithms
- Properties of an iterator correspond to properties of “container” for which it is defined

Input Iterators

- Operations
  - Equality, inequality
  - Next: ++j, j++
  - Dereference to get value: *j
- No guarantee can assign to *j
- Ex: istream_iterator<char>

Output Iterator

- Operations:
  - Dereference for assignment: *j = t
  - Next: ++j, j++
- May not have equality, inequality
- Ex: ostream_iterator<int>

Other Iterators

- Forward Iterators
  - Operations of both input and output iterator
  - Iterator value can be stored and used to traverse container
- Bidirectional Iterators
  - Operations of forward iterators
  - Previous: –j, –j
Random Access Iterators

- Bidirectional operators
- Addition, subtraction by integers: \( r+n \), \( r-n \)
- Jump by integer \( n \): \( r += n \), \( r -= n \)
- Iterator subtraction \( r - s \) yields integer
- Comparison of iterator values

Containers and Iterators

<table>
<thead>
<tr>
<th>Array, vector, deque</th>
<th>Random access</th>
</tr>
</thead>
<tbody>
<tr>
<td>List, set, multiset, map, multimap</td>
<td>bidirectional</td>
</tr>
</tbody>
</table>

Reverse Iterators

- Adapted from iterators of container classes
- Containers define the types
  - `reverse_iterator`
  - `const_reverse_iterator`
- Containers provide functions:
  - `rbegin()`
  - `rend()`

Choosing Container

- Vector used in place of dynamically allocated array
- List allows dynamically changing size for linear access
- Set used when need data kept sorted
- Map used when want indexed data
- Multi(set/map) when need multiple keys