CSCI 262
Data Structures

5 – Sets

SETS

Sets

- An abstract data type for holding unique elements
  - This is another container for objects
  - However, no repetition of objects
- Applications
  - Dictionary – a set of valid words
  - Compilers – sets of vars, classes, etc.
  - Spam filters – probability measure on sets of words in emails

The Set ADT

Sets do the following very efficiently:
- Find (test for presence of) an item in the set
- Insert an item into the set (preserving uniqueness)
- Remove an item from the set

Types of Sets

- Ordered sets
  - Items must be comparable
  - Items are iterated in sort order
  - Typically implemented using binary search trees – a future lecture topic
- Unordered sets
  - Typically faster than ordered sets
  - Items are iterated in no particular order
  - Implemented using hashtables – a future lecture topic
**STL Sets (Ordered)**

```cpp
#include <set>  
template <class T> class set
Methods:  
  find(T &val) // Find matching element (returns iterator)  
  count(T &val) // Count matching elements (returns count)  
  insert(T &val) // Insert element if not present  
  emplace(T &val) // Insert or return insertion iterator if present  
  erase(T &val) // Remove element if present  
  empty() // Return true if no elements  
  begin() // Get iterator to "first" element of set  
  end() // Get iterator marking "end" of set
```

**Set Example 1**

```cpp
set<string> words;  
string tt = "How much wood could a woodchuck...";  
istringstream s_in(tt);  
while (!s_in.eof()) {  
  string w;  
  s_in >> w;  
  words.insert(w);  
}  
for (string word: words)  
  cout << word << ' ';  

Output:  
How a chuck could if much wood woodchuck
```

**Set Example 2**

```cpp
set<string> shakespeare_words;  
ifstream fin("Shakespeare.txt");  
while (!fin.eof()) {  
  string w;  
  fin >> w;  
  shakespeare_words.insert(w);  
}  
while (true) {  
  string s;  
  cout << "Enter a word: ";  
  cin >> s;  
  if (shakespeare_words.count(s) > 0) {  
    cout << s << " is a Shakespeare word." << endl;  
  } else {  
    cout << s << " is not a Shakespeare word." << endl;  
  }  
}  
```

**Set Example 3**

```cpp
set<int> a = {1, 3, 7, 23, 2};  
set<int> b = {42, 17, 23, 3};  
set<int> a_intersect_b = a;  
set<int> a_union_b = b;  
set<int> a_minus_b = a;  
for (int x: a) {  
  a_union_b.insert(x);  
  if (b.count(x) > 0) {  
    a_minus_b.erase(x);  
  } else {  
    a_intersect_b.erase(x);  
  }  
}
```

**Iterators**

- Objects which point to elements of a container
  - In the abstract, work much like pointers
    - Use dereference operator (*) to access value
    - Use ++ to advance to next element
- Many types of iterators, different capabilities
  - Forward-only, bi-directional, random-access
  - Read-only or mutable
- The magic behind range-based for loops

**Iterators**

**Interlude**

**ITERATORS**
Obtaining Iterators

- All *iterable containers* provide:
  - `begin()` – returns iterator to “first” element
  - `end()` – returns iterator indicating “end”
  - N.b.: end() is a marker that does not point to a valid element: you cannot/should not dereference it!
- Container methods may return iterators
  - E.g., `find()`, but also `insert()`
- Standard library functions `find()`, `find_if()`, etc.

Iterating on Sets with Iterators

**Example 1**
```cpp
class string {...
    friend istream& operator >> (istream& s, string& t) { s >> t; return s; }
    friend ostream& operator << (ostream& s, const string& t) { return s << t; }
};

set<string> fruit = {"pear", "apple", "orange", "cherry"};

set<string>::iterator iter = fruit.begin();
while (iter != fruit.end()) {
    cout << *iter << ' ';
    iter++;
}
```

**Output for both is:**
```
apple cherry orange pear
```

Note sorted order!

**Example 2**
```cpp
class string {...
    friend istream& operator >> (istream& s, string& t) { s >> t; return s; }
    friend ostream& operator << (ostream& s, const string& t) { return s << t; }
};

set<string> fruit = {"pear", "apple", "orange", "cherry"};
for (auto iter = fruit.begin(); iter != fruit.end(); iter++) {
    cout << *iter << ' ';
}
```

**Range-based For Loop**

The range-based for loop works any *iterable container*:
```cpp
class string {...
    friend istream& operator >> (istream& s, string& t) { s >> t; return s; }
    friend ostream& operator << (ostream& s, const string& t) { return s << t; }
};

set<string> fruit = {"pear", "apple", "orange", "cherry"};
for (string f: fruit) {
    cout << f << " ";
}
```

Output is same as if we used iterators directly:
```
apple cherry orange pear
```

This is the easiest way to loop on a set.

STL Unordered Sets

- Same basic interface as (ordered) sets
- Iterable, but not in sort order:
  ```cpp
  #include <unordered_set>
  unordered_set<string> words;
  string tt = "How much wood could a woodchuck...";
  istringstream s_in(tt);
  while (!s_in.eof()) {
      string w;
      s_in >> w;
      words.insert(w);
  } for (string word: words) cout << word << ' ';
  ```

Output:
```
chuck a How much woodchuck could if wood
```

FINAL WORDS

Up Next

- Friday, Jan. 26
  - Lab 3 – Queues
  - Project 1 – Image Editor due
  - APT 2 assigned
- Monday, Jan. 29
  - Lecture: Maps
  - Lab 3 due
- Wednesday, Jan. 31
  - Debugging
  - Reading: Chapter 12.1