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

Set Operations

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

We also need a way to iterate (loop) over the elements in a 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>  // for unordered, use <unordered_set>
template <class T> class set  // for unordered, use unordered_set

Methods:
find(T &val)  // Find matching element (returns iterator)
count(T &val)  // Count matching element (returns count)
insert(T &val)  // Insert element, if not present
emplace(T &val)  // Same as insert
erase(T &val)  // Remove element, if present
size()  // Return number of elements
empty()  // Return true if no elements
begin()  // Get iterator to "first" element of set
end()  // Get iterator marking "end" of set
```

Iterating on Sets

- Not obvious how from documentation
- Need to know about iterators
  - Special objects implemented by some collections
  - While not part of C++ language, a crucial part of C++ standard library
- See section on iterators in a few slides
- Easiest way is to use the range-based for loop
  - Briefly mentioned in review lecture
  - Introduce by example in next slide, more soon

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

Sorted order because using ordered set

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
// an only slightly inefficient Sieve of Eratosthenes
vector<int> primes;
int n = 100;
// add all potential primes up to n
for (int j = 2; j <= n; j++) {
  primes.insert(j);
}
// remove all multiples of primes up to n, leaving only primes
for (int j = 2; j <= n; j++) {
  if (primes.count(j) > 0) {
    for (int k = j; k <= n; k += j) {
      primes.erase(k);
    }
  }
}
for (int x: primes) cout << x << ' ';  
```

Iterators

Interlude
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

Obtaining Iterators

- All iterable containers† provide:
  - begin() – returns iterator to “first” element
  - end() – returns iterator indicating “past the 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
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

Example 2
```cpp
set<string> fruit = {"pear", "apple", "orange", "cherry"};
for (auto iter = fruit.begin(); iter != fruit.end(); iter++) {
    cout << *iter << ' ';
}
```

auto

- auto is a C++ keyword
  - Use in place of specific type, if type can be inferred by compiler
  - Great for long, complicated type names (such as `set<string>::iterator` and worse)
  - Great in settings where knowing the type is not terribly important/far reaching (range-based loops)
  - Avoid casual use – e.g.:
    ```cpp
    auto x = 4;
    ```

Range-based For Loop

The range-based for loop works on any iterable container:
```cpp
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.
Discussion

We could use vectors as sets.

How might you implement:
- `insert()` (remember to ensure uniqueness)
- `count()` or `find()`
- `erase()`

What are the Big O costs of each?
What if we use a linked list instead?

We Can Do Better

We will see that:
- Hashtables (unordered sets) have expected O(1) cost for insert, find, erase
- Binary search trees (ordered sets) have worst-case O(log n) cost for insert, find, erase

Also, both do iteration of n elements in O(n) time.

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

- Read Chapter 15.1 – 15.2
- Wednesday, February 27
  - Maps
  - Reading: Chapter 15.3
- Friday, March 1
  - Lab 7 – TBA
  - Project 3 assigned