Sets & Multisets

- provide fast retrieval of elements (keys)
- elements must be unique in sets; multisets allow duplicate elements
- elements are ordered by ‘less-than” by default
- both classes have no mutable iterator type: don’t change the value of an element directly; remove it & insert a new one instead
- provide special search functions:
  - `find(elem)` returns the position of the first element equivalent to `elem` or `end()`
  - `lower_bound(elem)` returns the position of the first element not less than `elem`
  - `upper_bound(elem)` returns the position of the first element greater than `elem`
  - `equal_range(elem)` returns a `pair`
    - whose `first` is `lower_bound(elem)`
    - whose `second` is `upper_bound(elem)`

    (it basically returns a range of elements equivalent to `elem`)

    if `equal_range(elem).first==equal_range(elem).second`, `elem` is not found
  - `count(elem)` returns the number of elements equivalent to `elem`

```
#include <iostream>
#include <set>
using std::multiset;
using std::cout;
using std::endl;

int main() {
  multiset<int>  s;

  s.insert(2);
  s.insert(1);
  s.insert(2);
  s.insert(3);
  s.insert(5);
  s.insert(2);
  s.insert(5);
  cout << s.count(2) << endl;  // print: 3

  // note syntax; print: 5,5
  cout << *s.lower_bound(4) << ","
       << *s.upper_bound(4) << endl;

  // print: 3,5
  cout << *s.equal_range(3).first << ","
       << *s.equal_range(3).second << endl;

  s.erase(2);  // remove all 2s; returns number of
               //   elements removed
  // print: 1 3 5 5
  multiset<int>::iterator it;
  for (it = s.begin(); it != s.end(); ++it)
    cout << *it << " ";
  cout << endl;
}
```

- all standard associative containers have a member function `insert` that takes 2 iterators to specify a range of values to insert; this function returns nothing
- for `multiset` (& `multimap`), `insert(elem)` returns an iterator pointing to the newly-inserted element
- for `set` (& `map`), `insert(elem)` returns a pair whose
  - `first` is an iterator pointing an element in the container equivalent to `elem`
  - `second` is a boolean value — it is true if and only if `elem` is actually inserted into the set (i.e. an equivalent element was not in the set before)
- standard associative containers support bidirectional iterators

Maps

- provide fast retrieval of objects (values) based on keys
- keys must be unique

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main() {
  map<string, string>  phonebook;

  phonebook["jason"] = "123-4567";
  phonebook["stephen"] = "123-5678";
  // etc
  string name;
  while (cin >> name) {
    if (phonebook.find(name) != phonebook.end())
      cout << phonebook[name] << endl;
    else
      cout << "can’t find " << name << endl;
  }
}
```

  – note that a `map` is ordered by the “less-than operator” of the key by default
  – to create a `map` to store exam scores, we could use `map<string, int> scores`; in this case, the name is the key & the exam score is the value

– an iterator can be used to go thru a `map`; in the example above, we would use something like:

```
map<string, string>::iterator  it;
for ( it = phonebook.begin(); it != phonebook.end();
      ++it)
  cout << it->first << "," << it->second << endl;
```

The only thing new here is that we need to use the `first` & `second` members to access the key & value respectively. (A `map` essentially stores `pair`s.)

– note that in the `phonebook` example, the line

```
  phonebook["jason"] = "123-4567";
```

first initializes jason's phone to the default string (using the default ctor of `string`) before `"123-4567"` is assigned to it; for built-in arithmetic types, 0 is used as the default value

However, if an equivalent key is already in the map, the code changes the corresponding value.

– a "better" way to insert the key/value pair is

```
phonebook.insert(map<string, string>::
                 value_type("jason", "123-4567"));
```

Note however that this may fail if an element with an equivalent key is already in the map. (See `insert` for `set`.)

It is also convenient to use a `typedef`:

```
typedef map<string, string>::value_type  val_type;
```

– another alternative is to use:

```
phonebook.insert(make_pair<string, string>(
                     "jason", "123-4567"));
```

Multimaps

- provide fast retrieval of objects (values) based on keys
- allow duplicate keys

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main() {
  multimap<string, string>  phonebook;

  phonebook.insert(make_pair<string, string>(
                   "stephen", "123-5678"));
  phonebook.insert(make_pair<string, string>(
                   "albert", "123-6789"));
  phonebook.insert(multimap<string, string>::
                    value_type("albert", "123-0000"));
  // etc

  string name;
  multimap<string, string>::iterator  it;
  while (cin >> name) {
    for ( it = phonebook.lower_bound(name);
          it != phonebook.upper_bound(name); ++it)
      cout << it->second << endl;
  }
}
```

- as with `map`, & similar to `set` & `multiset`
  - `lower_bound(a_key)` returns the position of the first element whose key is not less than `a_key`; if there are no such keys, it returns `end()`
  - `upper_bound(a_key)` returns the position of the first element whose key is greater than `a_key`; if there are no such keys, it returns `end()`
  - `equal_range(a_key)` returns a `pair`
    * whose `first` is `lower_bound(a_key)`
    * whose `second` is `upper_bound(a_key)`

  Exercise: Change the program so that it prints a message when the name entered is not in the multimap