

Searching Multimaps

Lower and upper bounds Example

```
multimap<string, int> scores;  
scores.insert( {"Graham", 78} );  
scores.insert( {"Grace", 66} );  
scores.insert( {"Graham", 66} );  
scores.insert( {"Hareesh", 77} );  
scores.insert( {"Graham", 66} );
```

```
// Create an instance of std::multimap  
// Add some elements to it
```

```
// Elements are in the order "Grace", "Graham", "Graham", "Graham", "Hareesh"  
scores.lower_bound("Gordon");  
scores.upper_bound("Gordon");  
scores.lower_bound("Graham");  
scores.upper_bound("Graham");
```

```
// Returns an iterator to "Grace"  
// Returns an iterator to "Grace"  
// Returns an iterator to "Graham"  
// Returns an iterator to "Hareesh"
```

Finding an element using bounds

```
auto start = scores.lower_bound("Graham"); // Find first element with key "Graham"
auto finish = scores.upper_bound("Graham"); // Find first element after "Graham"

for (auto it = start; it != finish; ++it) { // Loop over the matching elements
    if (it->second == 66) {
        cout << "Found an element with key Graham and value 66!" << endl;
        break;
    }
}
```

equal_range()

- `equal_range()` is equivalent to calling `lower_bound()` followed by `upper_bound()`
 - It returns both the range iterators in a pair
 - The first member of the pair is the start and the second is the finish

```
auto eq = scores.equal_range("Graham");    // Range of elements with key "Graham"
```

```
for (auto it = eq.first; it != eq.second; ++it) ....
```

- We can use the returned iterator range with generic algorithms

```
// Call find_if() with a lambda expression to search for an element with value 66
```

```
auto result = find_if(start, finish,  
                      [](pair<string, int> p) { return p.second == 66; } );
```

```
if (result != finish) {                                // Did we find it?  
    cout << "Found an element with key Graham and value 66!" << endl;  
}
```

Finding repeated elements

- If we expect to find more than one element, we can search again, starting from the latest result

```
vector<pair<string, int>> results;           // vector to store search results

auto res = find_if(start, finish, [](pair<string, int> p) { return p.second == 66; } );

while (res != finish) {                     // Did we find it?
    results.push_back(*res);                // Add it to vector

    ++res;                                  // Move to next element and start another search
    res = find_if(res, finish, [](pair<string, int> p) { return p.second == 66; } );
}
```

Finding repeated elements (2)

- A more concise solution is to call an algorithm

```
vector<pair<string, int>> results;           // vector to store search results

copy_if(start, finish,
        back_inserter(results),
        [](pair<string, int> p) { return p.second == 66;}
);
```

- There are generic versions of these functions
- They can be called on any container
- They assume the containers are sorted

```
vector<int> vec {3, 1, 4, 1, 5, 9};  
sort(vec.begin(), vec.end());  
auto v = lower_bound(vec.begin(), vec.end(), 2);  
cout << *v << endl;
```

- There are also versions with a predicate which is called instead of the key's < operator