Set and MultiSet

- At the end of this topic, you will be able perform the following on Sets and MultiSets
- Understand when to use Sets, MultiSets, Maps and MultiMaps
- Understand when to use the Ordered vs UnOrdered versions
- Understand how to use the mapping of values to keys
- Be able to use a comparator function to perform ordering of keys
- Be able to perform the following on Sets: count, insert, find, copy, lower_bound, upper_bound, equal_range
- Return a pair or a tuple

- The *associative containers* provide *direct access* to store and retrieve elements via keys (often called search keys)
- Four *ordered associative containers*
  - `Multiset`
  - `Set`
  - `Multimap`
  - `Map`
    - Each of these maintains its keys in sorted order

- Four corresponding *unordered associative* `containers`
  - `unordered_multiset`
  - `unordered_set`
  - `unordered_multimap`
  - `unordered_map`
    - Offer the most of the same capabilities as their ordered counterparts.
- The primary difference between the ordered and unordered associative containers
  - The unordered ones do not maintain their keys in sorted order

Performance Tip 15.10

The unordered associative containers might offer better performance for cases in which it's not necessary to maintain keys in sorted order.

- Iterating through an *ordered associative container*
  - Traverses it in the sort order for that container
- Classes `multiset` and `set` provide operations for manipulating sets of values where the values are the keys
  - There is not a separate value associated with each key
- The difference between a `multiset` and a `set`
  - A `multiset` allows duplicate keys and a `set` does not

- Classes `multimap` and `map`
  - Provide operations for manipulating values associated with keys
    - mapped values
- The primary difference between a `multimap` and a `map`
  - A `multimap` allows duplicate keys with associated values to be stored
  - A `map` allows only *unique keys* with associated values

- The `multiset` *ordered associative container* (from header `<set>`)
  - Provides fast storage and retrieval of keys and allows duplicate keys
- Ordering is determined by a comparator function object
  - For example, in an integer `multiset`, elements can be sorted in *ascending order* by ordering the keys with comparator function object `less<int>`
- The data type of the keys in all *ordered associative containers* must support comparison based on the comparator function object
  - Keys sorted with `less<T>` must support comparison with `operator<`.

- If the keys used in the *ordered associative containers* are of user-defined data types
  - Those types must supply the appropriate comparison operators
- A `multiset` supports *bidirectional iterators*
  - But not *random-access iterators*
- If the order of the keys is not important
  - You can use `unordered_multiset` (header `<unordered_set>`) instead

- Figure 15.15 demonstrates the `multiset` *ordered associative container* for a `multiset` of `int`s with keys that are sorted in *ascending order*.

```
multiset< int, less< int > > intMultiset;
```

- Creates a `multiset` of `int`s ordered in *ascending order*, using the function object less<int>
- *Ascending order* is the default for a `multiset`, so `less<int>` can be omitted

- C++11 fixes a compiler issue with spacing between the closing > of `less<int>` and the closing > of the `multiset` type
  - Before C++11, if you specified this multiset's type as

```
multiset<int, less<int>> intMultiset;
```

  - The compiler would treat >> at the end of the type as the >> operator and generate a compilation error.
  - For this reason, you were required to put a space between the closing > of `less<int>` and the closing > of the `multiset` type (or any other similar template type, such as `vector<vector<int>>`)
  - As of C++11, the preceding declaration compiles correctly

```
intMultiset.count( 15 )
```

- `count` (available to all *associative containers*)
  - Counts the number of occurrences of the value `15` currently in the `multiset`

```
intMultiset.insert( 15 );
```

- Adds the value `15` to the `multiset`
- A second version of `insert` takes an iterator and a value as arguments and begins the search for the insertion point from the iterator position specified
- A third version of `insert` takes two iterators as arguments that specify a range of values to add to the `multiset` from another container

```
auto result = intMultiset.find(15);
```

- Locates the value `15` in the `multiset`
- Returns an `iterator` or a `const_iterator` pointing to the earliest location at which the value is found
- If the value is *not* found
  - Returns an `iterator` or a `const_iterator` equal to the value returned by calling `end` on the container

```
intMultiset.insert(a.cbegin(), a.cend());
```

- Function `insert` inserts the elements of array `a` into the `multiset`

```
copy(intMultiset.cbegin(),
     intMultiset.cend(),
     output);
```

- Copies the elements of the `multiset` to the standard output in *ascending order*

```
cout << "\n\nLower bound of 22: "
     << *( intMultiset.lower_bound( 22 ) );
cout << "\nUpper bound of 22: "
     << *( intMultiset.upper_bound( 22 ) );
```

- Use functions `lower_bound` and `upper_bound` (available in all *associative containers*)
  - To locate the earliest occurrence of the value `22` in the `multiset` and the element *after* the last occurrence of the value `22` in the `multiset`
- Both functions return `iterator`s or `const_iterator`s pointing to the appropriate location or the iterator returned by `end` if the value is not in the `multiset`

```
auto p = intMultiset.equal_range( 22 );
```

- Creates and intializes a `pair` object called `p`
- `auto` keyword infers the variable's type from its initializer
  - The return value of `multiset` member function `equal_range`, which is a `pair` object
  - Associate pairs of values
  - The contents of a `p` will be two `const_iterators` for the `multiset` of `int`s

- The `multiset` function `equal_range`
  - Returns a pair containing the results of calling both `lower_bound` and `upper_bound`
- Type `pair` contains two `public` data members called `first` and `second`

```
auto p = intMultiset.equal_range( 22 );
```

- Uses function `equal_range` to determine the `lower_bound` and `upper_bound` of 22 in the `multiset`

```
cout << "\n\nequal_range of 22:"
     << "\n   Lower bound: "
     << *( p.first )
     << "\n   Upper bound: "
     << *( p.second );
```

- Uses `p.first` and `p.second` to access the `lower_bound` and `upper_bound`
- We *dereferenced* the iterators to output the values at the locations returned from `equal_range`
- Though we did not do so here
  - You should always ensure that the iterators returned by `lower_bound`, `upper_bound` and `equal_range` are not equal to the container's end iterator before dereferencing the iterators

- C++ also includes class template `tuple`
  - Which is similar to `pair`
  - But can hold any number of items of various types
- As of C++11, class template `tuple` has been reimplemented using *variadic templates*
  - Templates that can receive a *variable* number of arguments

- The `set` *associative container* (from header `<set>`)
  - Used for fast storage and retrieval of unique keys
  - Implementation of a `set` is identical to that of a `multiset`
    - Except that a `set` must have unique keys
- If an attempt is made to insert a *duplicate key* into a `set`
  - The duplicate is ignored
  - The intended mathematical behavior of a set
  - We do not identify it as a common programming error

- A `set` supports *bidirectional iterators* (but not *random-access iterators*)

- Figure 15.16 demonstrates a `set` of `doubles`

```
set<double, less< double>>
   doubleSet(a.begin(), a.end());
```

- Creates a `set` of `double`s ordered in *ascending order*, using the function object `less<double>`
- The constructor call takes all the elements in `array` and inserts them into the `set`

```
copy(doubleSet.begin(), doubleSet.end(),
output);
```

- Algorithm `copy` to output the contents of the `set`

`auto p = doubleSet.insert( 13.8 );`

- Defines and initializes a `pair` to store the result of a call to `set` function `insert`
- The `pair` returned consists of
  - A `const_iterator` pointing to the item in the `set` inserted
  - And a `bool` value indicating whether the item was inserted
    - `true` if the item was not in the `set`; `false` if it was

- Places the value `13.8` in the `set`
- The returned `pair`, `p`,
  - Contains an iterator `p.first` pointing to the value `13.8` in the `set`
  - And a `bool` value that is `true` because the value was inserted
-

```
p = doubleSet.insert( 9.5 );
```

- Attempts to insert `9.5`, which is already in the `set`
- `9.5` is not inserted again because `set`s don't allow duplicate keys.
- `p.first` in the returned `pair` points to the existing 9.5 in the `set`

- This topic, taught and demonstrated how to perform the following on Sets and MultiSets
- Understand when to use Sets, MultiSets, Maps and MultiMaps
- Understand when to use the Ordered vs UnOrdered versions
- Understand how to use the mapping of values to keys
- Be able to use a comparator function to perform ordering of keys
- Be able to perform the following on Sets: count, insert, find, copy, lower_bound, upper_bound, equal_range
- Return a pair or a tuple