Sets

This topic teaches the set algorithms by demonstrating the usage including includes, set_difference, set_intersection, set_symmetric_difference and set_union and explaining the algorithms

- Figure 16.10 demonstrates algorithms `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` and `set_union` for manipulating *sets of sorted values*.

```
if (includes(a1.cbegin(), a1.cend(),
             a2.cbegin(), a2.cend()))
   cout << "a1 includes a2";
else
   cout << "a1 does not include a2";
```

- Calls the `includes` algorithm which compares two sets of *sorted* values to determine whether *every* element of the second set is in the first set
  - If so, `includes` returns `true`
  - Otherwise, it returns `false`
- The first two iterator arguments must be at least *input iterators* and must describe the first set of values
- Consists of the elements from `a1.cbegin()` up to, but *not* including, `a1.cend()`

- The last two iterator arguments must be at least *input iterators* and must describe the second set of values
- In this example, the second set consists of the elements from `a2.cbegin()` up to, but *not* including, `a2.cend()`
- A second version of algorithm `includes` takes a fifth argument that is a *binary predicate function* indicating the order in which the elements were originally sorted
- The two sequences must be sorted using the *same comparison function*

```
auto result1 = set_difference
    (a1.cbegin(), a1.cend(),
     a2.cbegin(), a2.cend(),
     difference.begin());
```

- Uses the `set_difference` algorithm to find the elements from the first set of sorted values that are not in the second set of sorted values (both sets of values must be in *ascending order*)
- The elements that are *different* are copied into the fifth argument
  - In this case, the array `difference`
- The first two iterator arguments must be at least *input iterators* for the first set of values
- The next two iterator arguments must be at least *input iterators* for the second set of values
- The fifth argument must be at least an *output iterator* indicating where to store a copy of the values that are *different*

- The algorithm returns an *output iterator* positioned immediately after the last value copied into the set to which the fifth argument points
- A second version of `set_difference` takes a sixth argument that is a *binary predicate function* indicating the order in which the elements were *originally* sorted
- The two sequences must be sorted using the *same comparison function*

```
auto result2 = set_intersection
   (a1.cbegin(), a1.cend(),
    a2.cbegin(), a2.cend(),
    intersection.begin() );
```

- Uses the `set_intersection` algorithm to determine the elements from the first set of sorted values that *are* in the second set of sorted values
  - Both sets of values must be in *ascending* order
- The elements *common to both sets* are copied into the fifth argument (in this case, array `intersection`)
- The first two iterator arguments must be at least *input iterators* for the first set of values

- The next two iterator arguments must be at least *input iterators* for the second set of values
- The fifth argument must be at least an *output iterator* indicating where to store a copy of the values that are the same
- The algorithm returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points
- A second version of `set_intersection` takes a sixth argument that is a *binary predicate function* indicating the order in which the elements were *originally* sorted
- The two sequences must be sorted using the *same comparison function*

```
auto result3 = set_symmetric_difference
   (a1.cbegin(), a1.cend(),
    a3.cbegin(), a3.cend(),
    symmetric_difference.begin());
```

- Uses the `set_symmetric_difference` algorithm to determine the elements in the first set that are *not* in the second set and the elements in the second set that are not in the first set
  - Both sets must be in *ascending order*
- The elements that are *different* are copied from both sets into the fifth argument (the array `symmetric_difference`)

- The first two iterator arguments must be at least *input iterators* for the first set of values
- The next two iterator arguments must be at least *input iterators* for the second set of values
- The fifth argument must be at least an *output iterator* indicating where to store a copy of the values that are different
- The algorithm returns an *output iterator* positioned immediately after the *last* value copied into the set to which the fifth argument points
- A second version of `set_symmetric_difference` takes a sixth argument that is a *binary predicate function* indicating the order in which the elements were originally sorted
- The two sequences must be sorted using the *same comparison function*

```
auto result4 = set_union
   (a1.cbegin(), a1.cend(),
    a3.cbegin(), a3.cend(),
    unionSet.begin());
```

- Uses the `set_union` algorithm to create a set of all the elements that are in *either or both* of the two sorted sets
  - Both sets of values must be in *ascending order*
- The elements are copied from both sets into the fifth argument (in this case the array `unionSet`)
- Elements that appear in both sets are only copied from the first set.
- The first two iterator arguments must be at least *input iterators* for the first set of values

- The next two iterator arguments must be at least *input iterators* for the second set of values
- The fifth argument must be at least an *output iterator* indicating where to store the copied elements
- The algorithm returns an *output iterator* positioned immediately after the last value copied into the set to which the fifth argument points
- A second version of `set_union` takes a sixth argument that is a binary predicate function indicating the order in which the elements were *originally sorted*
- The two sequences must be sorted using the *same comparison function*

This topic taught the set algorithms by demonstrating the usage including includes, set_difference, set_intersection, set_symmetric_difference and set_union and explaining the algorithms