Copy

This topic teaches the copy algorithm by demonstrating the usage including copy_backward, merge, unique and reverse and explaining the algorithms

- Figure 16.8 demonstrates algorithms `copy_backward`, `merge`, `unique` and `reverse`.

```
copy_backward(a1.cbegin(), a1.cend(),
              results.end());
```

- Uses the `copy_backward` algorithm to copy elements in the range from `a1.cbegin()` up to, but *not* including, `a1.cend()`
  - Placing the elements in `results` by starting from the element before `results.end()`
  - Working toward the beginning of the `array`
- Returns an iterator positioned at the *last* element copied into the `results` (i.e., the beginning of `results`, because of the backward copy)
- The elements are placed in `results` in the same order as `a1`

- `copy_backward` *can manipulate overlapping ranges of elements in a container as long as the first element to copy is not in the destination range of elementskward* through a sequence, respectively
- One difference between `copy_backward` and `copy` is that
  - The iterator returned from `copy` is positioned after the last element copied
  - The one returned from `copy_backward` is positioned *at* the last element copied (i.e., the first element in the sequence)
- `copy_backward` *can* manipulate *overlapping* ranges of elements in a container as long as the first element to copy is *not* in the destination range of elements

- In addition to the `copy` and `copy_backward` algorithms
  - C++11 now includes the move and `move_backward` algorithms

```
merge(a1.cbegin(), a1.cend(),
      a2.cbegin(), a2.cend(),
      results2.begin() );
```

- Uses the `merge` algorithm to combine two *sorted ascending sequences* of values into a third sorted ascending sequence
- Requires five iterator arguments
- The first four must be at least *input iterators*
- The last must be at least an *output iterator*
- The first two arguments specify the range of elements in the first sorted sequence (`a1`), the second two arguments specify the range of elements in the second sorted sequence (`a2`) and the last argument specifies the starting location in the third sequence (`results2`) where the elements will be merged
- A second version of this algorithm takes as its sixth argument a *binary predicate function* that specifies the *sorting order*

```
array< int, SIZE + SIZE > results2;
```

- Creates the `array` `results2` with the number of elements in `a1` and `a2` (SIZE)
- Using the `merge` algorithm requires that the sequence where the results are stored be at least the size of the sequences being merged
- If you do not want to allocate the number of elements for the resulting sequence before the `merge` operation
  - You can use the following statements:

```
vector< int > results2;
merge(a1.begin(), a1.end(),
      a2.begin(), a2.end(),
      back_inserter( results2 ) );
```

- The argument `back_inserter(results2)` uses function template `back_inserter` (header `<iterator>`) for the `vector` `results2`
- A `back_inserter` calls the container's default `push_back` function to insert an element at the end of the container
- If an element is inserted into a container that has no more space available, *the container grows in size*—which is why we used a `vector` in the preceding statements, because `array`s are fixed size
- Thus, the number of elements in the container does *not* have to be known in advance

- There are two other inserters
  - `front_inserter`
    - Uses `push_front` to insert an element at the *beginning* of a container specified as its argument)
  - `Inserter`
    - Uses `insert` to insert an element *at* the iterator supplied as its second argument in the container supplied as its first argument

```
auto endLocation =
    unique(results2.begin(), results2.end());
```

- Uses the `unique` algorithm on the *sorted* sequence of elements in the range from `results2.begin()` up to, but *not* including, `results2.end()`
- After this algorithm is applied to a sorted sequence with *duplicate* values, only a *single* copy of each value remains in the sequence
- Takes two arguments that must be at least *forward iterators*

- The algorithm returns an iterator positioned *after the last element* in the sequence of unique values
- The values of all elements in the container after the last unique value are *undefined*
- A second version of this algorithm takes as a third argument a *binary predicate function* specifying how to compare two elements for *equality*

```
reverse( a1.begin(), a1.end() );
```

- Uses the `reverse` algorithm to reverse all the elements in the range from `a1.begin()` up to, but *not* including, `a1.end()`
- Takes two arguments that must be at least *bidirectional iterators*

- `copy_if`
  - Copies each element from a range if the *unary predicate function* in its fourth argument returns `true` for that element
  - The iterators supplied as the first two arguments must be *input iterators*
  - The iterator supplied as the third argument must be an *output iterator* so that the element being copied can be assigned to the copy location
  - Returns an iterator positioned after the *last* element copied

- `copy_n`
  - Copies the number of elements specified by its second argument from the location specified by its first argument (an *input iterator*)
  - The elements are output to the location specified by its third argument (an *output iterator*)

This topic taught the copy algorithm by demonstrating the usage including copy_backward, merge, unique and reverse and explaining the algorithms