Math Algorithms

This topic teaches the math algorithms by demonstrating the usage including `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `minmax_element`, `for_each` and `transform`. and explaining the algorithms

- Figure 16.5 demonstrates several common mathematical algorithms, including `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `minmax_element`, `for_each` and `transform`.

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

- Uses the `random_shuffle` algorithm to reorder randomly the elements in the range `a1.begin()` up to, but not including, `a1.end()`
- Takes *two random-access iterator* arguments. This version of `random_shuffle` uses rand for randomization and produces the same results each time you run the program unless you seed the random-number generator with `srand`
- Another version of `random_shuffle` receives as its third argument a C++11 uniform random-number generator

```
int result =
    count(a2.cbegin(), a2.cend(), 8);
```

- Uses the `count` algorithm to count the elements with the value `8` in the range `a2.cbegin()` up to, but *not* including, `a2.cend()`
- Requires its two iterator arguments to be at least *input iterators*

```
result = count_if
    (a2.cbegin(), a2.cend(), greater9);
```

- Uses the `count_if` algorithm to count elements in the range from `a2.cbegin()` up to, but *not* including, `a2.cend()` for which the *predicate function* `greater9` returns `true`
- Requires its two iterator arguments to be at least *input iterators*

```
cout
    << "\n\nMinimum element in a2 is: "
    << *(min_element(a2.cbegin(), a2.cend()));
```

- Uses the `min_element` algorithm to locate the *smallest* element in the range from `a2.cbegin()` up to, but *not* including, `a2.cend()`.
- Returns a *forward iterator*
  - Located at the *first* smallest element
  - Or `a2.end()` if the range is *empty*
- The algorithm's two iterator arguments must be at least *forward iterators*
- A second version of this algorithm takes as its third argument a binary function that compares two elements in the sequence
  - Returns the `bool` value `true` if the first argument is *less than* the second

Error-Prevention Tip 16.1

It's a good practice to check that the range specified in a call to `min_element` is not empty and that the return value is not the "past the end" iterator.

```
cout
   << "\nMaximum element in a2 is: "
   << *(max_element(a2.cbegin(),a2.cend()));
```

- Uses the `max_element` algorithm to locate the largest element in the range from `a2.cbegin()` up to, but *not* including, `a2.cend()`
- The algorithm returns a *forward iterator* located at the *first* largest element
- The algorithm's two iterator arguments must be at least *forward iterators*
- A second version of this algorithm takes as its third argument a *binary predicate function* that compares the elements in the sequence
  - Takes two arguments and returns the `bool` value `true` if the first argument is *less than* the second

```
auto minAndMax =
   minmax_element(a2.cbegin(), a2.cend());
```

- Uses the new `minmax_element` algorithm to locate both the *smallest* and *largest* elements in the range from `a2.cbegin()` up to, but *not* including, `a2.cend()`
- Returns a pair of forward iterators located at the smallest and largest elements, respectively
- If there are duplicate smallest or largest elements
  - The iterators are located at the first smallest and last largest values
- The algorithm's two iterator arguments must be at least *forward iterators*
- A second version of this algorithm takes as its third argument a *binary predicate function* that compares the elements in the sequence.
  - Takes two arguments and returns the `bool` value `true` if the first argument is less than the second

```
cout
   << "The total of the elements in a1 is: "
   << accumulate( a1.cbegin(), a1.cend(), 0 );
```

- Uses the `accumulate` algorithm (in header `<numeric>`) to sum the values in the range from `a1.cbegin()` up to, but *not* including, `a1.cend()`
- The algorithm's two iterator arguments must be at least *input iterators* and its third argument represents the initial value of the total

- A second version of this algorithm takes as its fourth argument a general function that determines how elements are accumulated
- The general function must take *two* arguments and return a result
- The first argument to this function is the current value of the accumulation
- The second argument is the value of the current element in the sequence being accumulated

```
for_each(a1.cbegin(), a1.cend(),
         outputSquare );
```

- Uses the `for_each` algorithm to apply a general function to every element in the range from `a1.cbegin()` up to, but *not* including, `a1.cend()`
- The general function takes the current element as an argument and may modify that element
  - If it's received by reference and is not `const`
- Algorithm `for_each` requires its two iterator arguments to be at least *input iterators*

```
transform(a1.cbegin(), a1.cend(),
          cubes.begin(), calculateCube);
```

- Uses the `transform` algorithm to apply a general function to every element in the range from `a1.cbegin()` up to, but *not* including, `a1.cend()`
- The general function (the fourth argument) should take the current element as an argument, must *not* modify the element and should return the `transform`ed value
- Requires its first two iterator arguments to be at least input iterators and its third argument to be at least an *output iterator*
- The third argument specifies where the `transform`ed values should be placed.
  - Note that the third argument can equal the first

- Another version of `transform` accepts five arguments
  - The first two arguments are *input iterators* that specify a range of elements from one source container
  - The third argument is an *input iterator* that specifies the first element in another source container
  - The fourth argument is an *output iterator* that specifies where the transformed values should be placed
  - The last argument is a general function that takes two arguments
- This version of `transform`
  - Takes one element from each of the two input sources
  - Applies the general function to that pair of elements
  - Then places the transformed value at the location specified by the fourth argument

This topic taught math algorithms by demonstrating the usage including `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `minmax_element`, `for_each` and `transform`. and explaining the algorithms