1. Algorithms that take a single iterator denoting a second sequence assume that the second sequence is at least as large as the first.
2. The type of the third argument to **accumulate** determines which addition operator is used and is the type that accumulate returns.
3. Algorithms that write to a destination iterator assume the destination is large enough to hold the number of elements being written.
4. Lambda
   1. We can **omit** either or both **parameter list** and **return type** but must always include the capture list and function body
   2. Lambdas with function bodies that contains anything other than a single return statement that do not specify a return type return void.
   3. Unlike ordinary functions, a lambda may not have default arguments.
   4. A lambda may use a variable local to its surrounding function only if the lambda captures that variable in its capture list.
   5. The **capture list** is used for local nonstatic variables only; lambdas can use local statics and variables declared outside the function directly.
   6. When we define a lambda, the compiler generates a new(unnamed) class type that corresponds to that lambda. By default, the class generates from a lambda **contains a data member** corresponding to the variables **captured** by the lambda, which will be init **when a lambda obj is** **created**. (like the data member of any class)
   7. **Capture by Value or by Reference**
      1. When we **capture** a variable **by reference**, we must ensure that the variable exists at the time that the lambda executes.
      2. As a rule, we can avoid potential problem with captures by minimizing the data we capture. Moreover, if possible, avoid capturing pointers or references.
      3. use **= (by value)** or **& (by ref)** to use implicit captures
      4. When mix implicit and explicit captures, the first item in the capture list must be an **&** or **=**. The explicit vars must use the alternate form.
      5. Can’t capture **this** by reference
      6. By default, a lambda may not change the value of a variable that it copies by value. If we want to do that, follow the parameter list with the keyword **mutable**. (e.g. [ ] (int val) mutable -> int {return val \* val;})
      7. Lambdas that are mutable may not omit the parameter list.
5. Binding Arguments
   1. We can’t use bind directly to replace captures of reference, because **bind** copies its arguments and we cannot copy an ostream.
   2. If we want to pass an obj to **bind** without copying it, we must use the library **ref** function, which returns an object that contains the given reference and that is itself copyable.
   3. There is also a **cref** function that generates a class that holds a reference to **const**.
6. When we create an istream\_iterator, we can bind it to a stream. Alternatively, we can default initialize the iterator, which creates an iterator that we can use as the off-the-end value.
7. Call the **reverse\_iterator**’s **base** member, which gives us its corresponding ordinary iterator (which points to the position after the position reverse\_iterator points).
8. The fact that reverse iterators are intended to represent ranges and that these ranges are asymmetric has an important consequence: When we initialize or assign a reverse iterator from a plain iterator, the resulting iterator does not refer to the same element as the original. [Example](Generic%20Algorithms.cpp)
9. **Input iterator**. We are guaranteed that \*iter++ is valid but incrementing an input iterator may invalidate all other iterators into the stream. As a result, there is no guarantee that we can save the state of an input iterator and examine an element through that saved iterator. Input iterators, therefore, may be used only for single-pass algorithms.
10. **Output iterator.** Like input iterators, output iterators may be used only for single-pass algorithms.
11. Container-Specific Algorithms
    1. Unlike the other containers, list and forward\_list define several algorithms as members. In particular, the list types define their own versions of **sort**, **merge**, **remove**, **reverse**, and **unique**.
    2. A list can “swap” its elements by changing the links among it’s elements rather than swapping the value of those elements. As a result, the list-specific versions of these algorithms can achieve much better performance than the corresponding generic versions.
    3. The list member versions should be used in preference to the generic algorithms for **list**s and **forward\_list**s
    4. list versions of algorithms change the underlying container.
    5. For example, the list version of remove removes the indicated elements. The list version of unique removes the second and subsequent duplicate elements.