【C++】Day43(2)

Class	C++
Date	@January 22, 2022
Material	
Series Number	
Summary	

【Ch10】Generic Algorithm

10.5.2 Algorithm Parameter Patterns

Superimposed on any other classification of the algorithms is a set of parameter conventions. Most of algorithms have one of the following four forms:

```
alg(beg, end, other args);
alg(beg, end, dest, other args);
alg(beg, end, beg2, other args);
alg(beg, end, beg2, end2, other args);
```

where `alg` is the name of the algorithm, and `begin` and `end` denote the input range which the algorithm operates.

Although nearly all algorithms take an input range, the presence of the other parameters depends on the work begin performed.

Algorithms with a Single Destination Iterator

A `dest` parameter is an iterator that denotes a destination in which the algorithm can write its output. Algorithms assume that it is safe to write as many elements as needed.

Warning: Algorithms that write to an output iterator assume the destination is large enough to hold the output

If `dest` is an iterator that refers directly to a container, then the algorithm writes its output to existing elements within the container.

More commonly, `dest` is bound to an insert iterator or an `ostream_iterator`. An insert iterator adds new elements to the container, thereby ensuring that there is enough space. An `ostream_iterator` writes to an output stream, again presenting no problem regardless of how many elements are written.

Algorithms with a Second Input Sequence

Algorithms that take either `beg2` alone or `beg2` and `end2` use those iterators to denote a second input range. These algorithms typically use the elements from the second range in combination with the input range to perform a computation.

Algorithms that take only `beg2` treat `beg2` as the first element in the second input range. The end of this range is not specified. Instead, these algorithms assume that the range starting at `beg2` is at least as large as the one denoted by `beg`, `end`.

Warning: Algorithms that take beg2 alone assume that the sequence beginning at beg2 is as large as the range denoted by beg and end.

10.5.3 Algorithm Naming Conventions

Separate from the parameter conventions, the algorithms also conform to a set of naming and overload conventions.

These conventions deal with how we supply an operation to use in place of the default `<` or `==` operator and with whether the algorithm writes to its input sequence or to a separate destination.

Some Algorithms Use Overloading to Pass a Predicate

Algorithms that take a predicate to use in place of the `<` or `==` operator, and that do not take other arguments, typically are overloaded.

One version of the function uses the element type’s operator to compare elements; the second takes an extra parameter that is a predicate to use in place of `<` or `==`:

```
unique(beg, end); //uses the == operator to compare the elements
unique(beg, end, comp); //uses comp to compare the elements
```

Because the two functions differ as to the number of arguments, there is no possible ambiguity as to which function is being called.

Algorithms with _if Versions

Algorithms that take an element value typically have a second named(not overloaded) version that takes a predicate in place of the value. The algorithms that take a predicate have the suffix `_if` appended:

```
find(beg, end, val);
find_if(beg, end, pred); //find the first instance for which pred is true
```

Distinguishing Versions that Copy from Those That Do Not

By default, algorithms that rearrange elements write the rearranged elements back into the given input range. These algorithms provide a second version that writes to a specified output destination. As we've seen, algorithms that write to a destination append `_copy` to their names:

```
revser(beg, end); //reverse the elements in the input range
reverse_copy(beg, end, dest); //copy elements in reverse order into dest
```

10.6 Container-Specific Algorithms

Unlike the other containers, `list` and `forward_list` define several algorithms as members.

This is because the generic version of sort, for example, requires random-access iterators but these types offer bidirectional and forward iterators, respectively.

These list-specific operations are listed below:

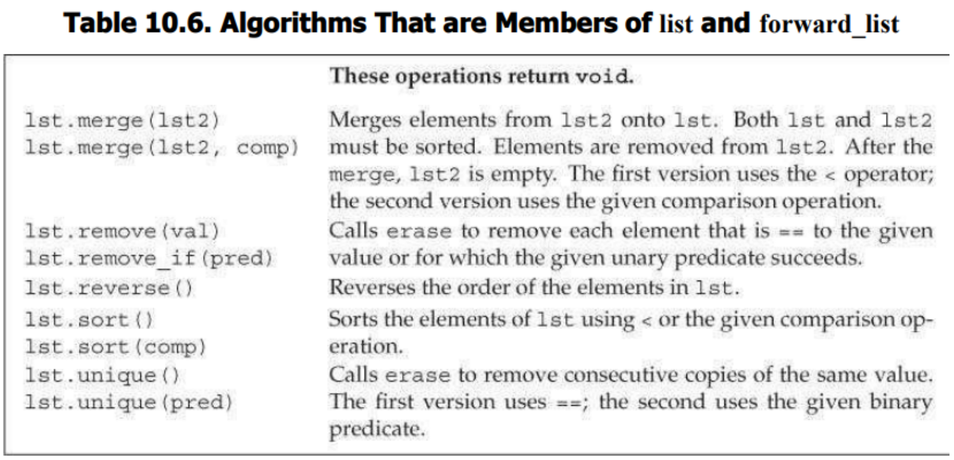


Table 10.6. Algorithms That are Members of list and forward_list

	These operations return `void`.

`lst.merge(lst2)`
`lst.merge(lst2, comp)`
	Merges elements from `lst2` onto `lst`. Both `lst` and `lst2` must be sorted. Elements are removed from `lst2`. After the `merge`, `lst2` is empty. The first version uses the < operator; the second version uses the given comparison operation.

`lst.remove(val)`
`lst.remove_if(pred)`
	Calls `erase` to remove each element that is == to the given value or for which the given unary predicate succeeds.

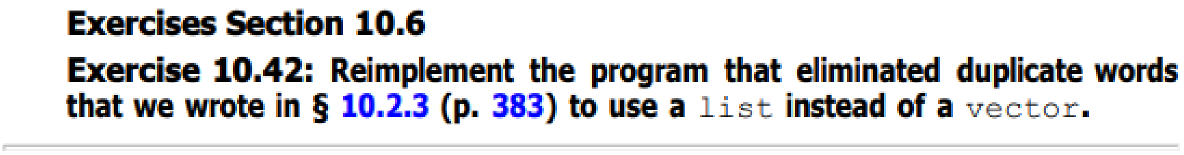
`lst.reverse()`	Reverses the order of the elements in `lst`.

`lst.sort()`
`lst.sort(comp)`
	Sorts the elements of `lst` using < or the given comparison operation.

`lst.unique()`
`lst.unique(pred)`
	Calls `erase` to remove consecutive copies of the same value. The first version uses ==; the second uses the given binary predicate.

Best Practices: The list member versions should be used in preference to the generic algorithms for lists and forward_lists.

Exercise



Exercises Section 10.6

Exercise 10.42: Reimplement the program that eliminated duplicate words that we wrote in § 10.2.3 (p. 383) to use a `list` instead of a `vector`.

See 10_42.cpp for code