Generic Algorithms

Roadmap

STL generic algorithms
Interaction between algorithms and iterators
Generic algorithm goals, documentation, and design
Sequences
Descriptions of some STL algorithms
A preference for algorithms

STL Generic Algorithms

The STL provides over seventy generic algorithms.

The generic algorithms are written as function templates.

STL algorithms are designed to work with iterators, and are therefore independent of STL containers.

- New generic algorithms can be added without updating existing containers.
- New containers can be added without updating algorithms.
- The algorithms may be employed without recourse to any container at all, if suitable iterators are available.

For all but the most trivial uses, we prefer to write non-trivial container access code as a generic algorithm in the STL style.

- This allows us to leverage our work without compromising efficiency.

Not as complex as it may appear at first.

- Many are arranged in families of similar functions.
- Many take pairs of iterators that define a range.
- If an algorithm returns an iterator, it is the same type as one of the argument iterators.
- The end-of-sequence iterator value is usually used to indicate failure.

Most of the generic algorithms are declared in <algorithm>, with a few numerical algorithms in <numeric>, and some associated class and function templates in <functional>.

As we saw in the previous section, we can define iterators for iostreams, allowing us to employ STL's generic algorithms on streams.

Algorithms and Iterators

A generic algorithm implicitly specifies the types of iterators it requires through the operations its implementation employs.

The following (almost correct) implementation of the count algorithm requires only an input iterator.

```
template <typename Iter, typename T>
int count( Iter first, Iter last, const T &value ) {
      int n = 0;
      for( Iter i = first; i != last; i++ )
            if( *i == value )
                  n++;
      return n;
}
```


This implicit specification of the properties required of types for template specialization, is, of course, true of any template. The requirements exist on two levels. First, the parameters supplied to a template during expansion must allow the compiler to generate legal C++ code; that is, the syntax and static semantics of the instantiated template must be legal C++. More difficult to guarantee is that the meaning of the instantiated template is appropriate, and that it matches the template designer's intent.

Gotcha: One area of particular concern is that of the interaction of instantiated templates with exception handling.

Note that most generic algorithms work with sequences of container elements that are defined by two iterators. The first iterator refers to the first element of the sequence, and the second refers to a point "one past" the end of the sequence.

Note: The count generic algorithm is actually declared to return an integral type that may be different from int. The return is shown here as int for simplicity. The actual return type is iterator_traits<Iter>::difference_type.

The iter_swap template is an STL generic algorithm that swaps two objects through iterators to the objects. The iterators do not have to be the same type.

```
template <class For1, class For2>
void iter_swap( For1 a, For2 b );
```

Implicit Iterator Requirements

However, the following implementation of the reverse algorithm requires a bidirectional iterator.

```
template <typename Iter>
void reverse( Iter first, Iter last ) {
      while( true )
            if( first == last || first == --last )
                  return;
            else
                  iter_swap( first++, last );
}
```

Algorithms, Iterators, and Documentation

Typically, and ideally, users of generic algorithms should not have to look at the algorithm implementations in order to use them properly.

Therefore, descriptive names are used in the algorithm template declarations.

- OutputIterator
- InputIterator
- ForwardIterator
- BidirectionalIterator
- RandomAccessIterator

Stroustrup suggests an alternative standard:

- Out
- In
- For
- Bi
- Ran

Gotcha: However, it is recommended, for reasons of documentation, that you stick with either the standard type names or with Stroustrup's. Don't make up your own.

Algorithms, Iterators, and Documentation

For example:

```
template <typename InputIterator, typename T>
int count( InputIterator first, InputIterator last, const T &value );

template <typename BidirectionalIterator>
void reverse( BidirectionalIterator first, BidirectionalIterator last );
```

If iterators of the same category are required from (potentially) different containers, an integer is appended to distinguish them.

```
template< typename In1, typename In2, typename Out>
Out merge( In1 first1, In1 last1,
        In2 first2, In2 last2,
        Out result );
```


The iterator type names listed in the first example are those used in the ANSI/ISO C++ standard. Bjarne Stroustrup recommends that shorter identifiers be used for readability: In, Out, For, Bi, and Ran. It's easy to see his point with the "standard" version of the merge declaration:

```
template< typename InputIterator1, typename InputIterator2, typename OutputIterator>
OutputIterator merge( InputIterator1 first1, InputIterator1 last1,
                      InputIterator2 first2, InputIterator2 last2,
                      OutputIterator result );
```

Different names are required for the two input iterator sequences because the iterator typenames specify only a minimum requirement for the iterators used in a template expansion. Therefore a merge of a list and a vector could be performed (using a bidirectional and random access iterators, respectively) or a merge of two lists could be performed (using all bidirectional iterators). Using the same name for all the iterators would require that all the iterators have the same type when the template is expanded.

However, it's a good idea to use the same typename for iterators from the same sequence, since this will help to catch sequence mismatch errors (for instance, inadvertently passing the start of a list and the end of a vector as a sequence).

Carelessly-Written Algorithms

Compare this version of count to our earlier version.

```
template <typename In, typename T>
int count( In first, In last, const T &value ) {
      int n = 0;
      for( In i = first; i < last; ++i )
            if( *i == value )
                  ++n;
      return n;
}
```

This version of the algorithm is no faster than the first, but it is less useful.

Generic algorithms should be written to the least powerful iterator for which an efficient implementation is possible.

In some cases, a particular STL implementation will provide multiple implementations of an algorithm for different iterator types, along with a compile-time mechanism for selecting the proper implementation. The technique employs "iterator tags" to determine the type of the iterator at compile time. This technique is discussed in the advanced section of this course.

For example, a somewhat simpler and faster version of the reverse algorithm can be produced for random access iterators.

```
template <typename Rand>
void reverse( Rand first, Rand last ) {
      while( first < last )
            iter_swap( first++, --last );
}
```

Note that it does not always make sense to code an algorithm to the most general iterator possible. For instance, it is possible to write a sort algorithm in terms of a forward or bidirectional iterator, but a much more efficient sort can be produced for a random access iterator. For this reason, STL's sort algorithm is defined to work with random access iterators only. (However, the list container has a member function that sorts the list.)

```
template <typename RandomAccessIterator>
void sort( RandomAccessIterator first, RandomAccessIterator last );
```

Note also that the documentation of the second version of count is incorrect, in that the type name of the argument iterator is In. However, this error will not be apparent until a user of the count template attempts to expand it with a non-random access iterator.

Sequence Errors

The compiler can check that the iterators that define a sequence are of the correct type.

```
vector<int> v1;
vector<double> v2;
sort( v1.begin(), v2.end() ); // error!
```

But it will not be able to detect many other sequence errors.

```
vector<int> v3;
sort( v1.begin(), v3.end() );
```

It is also often inconvenient to continually write c.begin(), c.end() to indicate all the elements of a container.

```
int num2s = count( v1.begin(), v1.end(), 2 );
```

One reasonable approach is to use the standard mechanisms and just be careful and diligent.

Another approach is to write helper templates for specific STL algorithms.

```
template <class C, typename T>
inline int OrgSemantics::count( const C &c, const T &val )
        { return std::count( c.begin(), c.end(), val ); }
```

Sorting

sort requires a sequence described by random access iterators.

```
vector<Member> &members = getMembers();
sort( members.begin(), members.end() );
```

Remember that sort uses < to compare sequence elements!

```
const char *a[] = { "Fred", "Ann", "Zoe" };
vector<string> b( a, a+3 );

sort( a, a+3 );
sort( b.begin(), b.end() );
for( int i = 0; i < 3; ++i )
   cout << a[i] << " : " << b[i] << endl;
```

We'll see how to modify this behavior later.

For these examples, assume a simple class Member with (at least) the following interface:

```
class Member {
  public:
        Member( const char *name = "" );
        ~Member(); // not a base class!
        Member( const Member & );
        Member &operator =( const Member & );
        friend bool operator ==( const Member &, const Member & );
        friend bool operator <( const Member &, const Member & );
        // ...
};
```

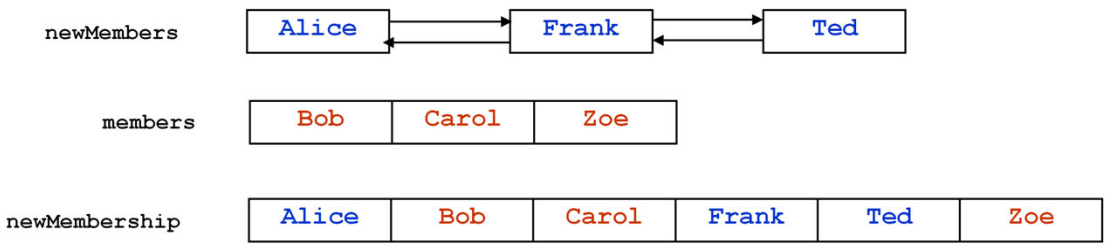
By default, sort uses the < operator for comparison. Later we'll see how to supply different comparison operators to sort.

sort is generally implemented as an introsort, which guarantees produces NlgN performance. sort may generally as a quicksort, which usually produces NlgN performance. Worse case behavior can be much slower, however. Using partial_sort guarantees NlgN behavior, but may be slower than sort in general.

Merging

merge merges two sorted sequences into an output sequence.

```
extern vector<Member> members;
list<Member> &newMembers = getNewMembers();
newMembers.sort();
vector<Member> newMembership;
merge( members.begin(), members.end(),
       newMembers.begin(), newMembers.end(),
       back_inserter( newMembership ) );
```



Note that the two sequences passed to merge have to be sorted, but they do not have to be described by random access iterators, only by input iterators.

Make sure that the two sequences passed to merge are sorted by the same criterion!

See also the inplace_merge algorithm.

The merge algorithm is linear in the number of elements (the sum of the numbers of elements in both ranges).

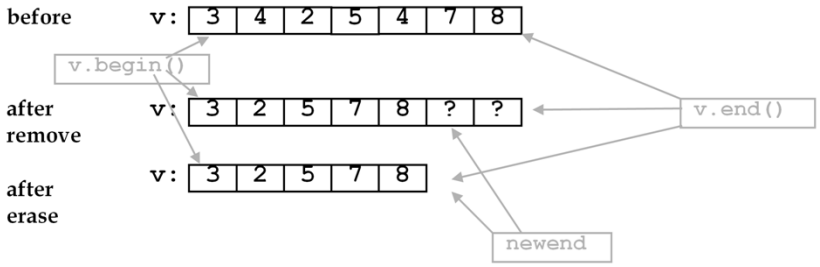
remove

remove "compresses out" elements with a given value.

```
int a[ ] = { 3, 4, 2, 5, 4, 7, 8 };
vector<int> v( a, a+7 );
vector<int>::iterator newend = remove( v.begin(), v.end(), 4 );
v.erase( newend, v.end() );
```

Note that remove can not modify the container, only the sequence of elements. remove can't change the size of the sequence.

Removed elements should be erased.



Gotcha: Remember that unique (and remove) can not change the size of a container (since they work with sequences, not containers), and so do not actually remove the elements; they are just shifted around.

Note that unique and remove can also be used on a predefined array.

```
int a[] = { 3, 4, 2, 5, 4, 7, 8 };
int *newend = remove( a, a+7, 4 );
```

Gotcha: Note that unique will not remove all duplicate elements unless all elements that compare equal are contiguous. Typically, the sequence passed to unique is sorted, but it does not have to be.

The Erase/Remove Idiom

It is much more typical to write erase and remove in the same expression.

```
v.erase( remove( v.begin(), v.end(), 4 ), v.end() );

newMembership.erase(unique( newMembership.begin(),
      newMembership.end() ), newMembership.end() );
```

This style may look more confusing than at first, but it's actually easier to read for an experienced STL programmer.

Finding

find is used to locate a value in a sequence, starting from the beginning of the sequence.

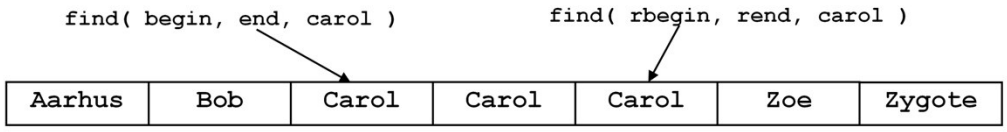
```
Member carol( "Carol" );
vector<Member>::iterator result
   = find( newMembership.begin(), newMembership.end(), carol );
```

Failure is indicated by the end value of the sequence.

```
if( result == newMembership.end() )
   cout << "Carol doesn't live here anymore." << endl;
```

A trick: You can find a last occurrence by using find with reverse iterators (and comparing against rend() instead of end()).

```
vector<Member>::reverse_iterator lresult
   = find( newMembership.rbegin(), newMembership.rend(), carol );
```



Correctness and Efficiency

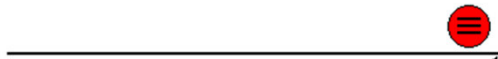
Often it's a mistake to write a hand-coded loop when a generic algorithm is available to do the same operation.

The generic algorithm is generally as efficient--and often more efficient--than a hand-coded loop.

- STL algorithms are written by experts whose job it is to shave nanoseconds from typical usages
- STL algorithms often employ metaprogramming techniques that are not practical for day-to-day loop writing

It is also more likely to be correct.

- off-by-one errors and the like are reduced
- as we'll see, it can be challenging to write loops that work on sequences from different STL containers!



STL Algorithms: Exercises

STL Algorithms Exercises

Exercise 1: directory Interleave

- Write a generic algorithm to interleave the content of two sequences to an output sequence. If one sequence is longer than the other, simply omit any additional elements.

Exercise 2: directory CopyImpl

- Implement the myCopy generic algorithm:

```
template <typename In, typename Out>
Out myCopy( In b, In e, Out dest );
```

Exercise 3: directory CopyMiddle

- Copy the names (strings) from cin to cout, but copy only the lexicographically middle third of the names. The result does not have to be sorted.

Exercise 4: directory CopySort

- Copy the names from cin to cout in sorted order, but remove duplicates.

STL Algorithms Exercises

Exercise 5: directory SlowSort

- Compare the relative speed of the slowSort algorithm and the std::sort algorithm.

Exercise 6: directory CrosswordSort

- Write a function object that compares two character strings first by length, then by dictionary (lexicographic) order within strings of the same length ("Zeno" < "Demosthenes" and "cat" < "dog").
- Read in an unsorted file of names from cin, sort it with your comparitor, and copy the names to cout.

Exercise 7: directory Binary

- Making liberal use of the binary search algorithms described in this section, implement the container whose interface is shown in the file assocvec.h. (Note that this container interface does not adhere to the STL conventions, and is therefore not as useful as an STL-compliant container.)
- Thought question: How does the performance of insertion compare to that of lookup and initialization of an AssocVec? When would you prefer an AssocVec to a standard set?

STL Algorithms Exercises

Exercise 8: directory Anagram

- An anagram is a word formed by the rearrangement of the letters of another word. For example "carthorse" is an anagram of "orchestra," and "act" is an anagram of "cat."
- Write an interactive program to find all the anagrams of a word input by the user. To determine whether a particular permutation of the input word is a "real" word, consult the file diction, which is a dictionary of approximately 20,000 words, one word per line.
- Don't be too concerned about developing a perfect user interface; anything that works is fine.
- Both the idea for the anagram program and the content of the dictionary come from David Musser & Atul Saini, *STL Tutorial and Reference Guide*, Addison Wesley, 1996.