STL Generic Algorithms

STL Generic Algorithms

- Algorithms operate over *iterators* rather than containers

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

vector<int> v1 {1, 2, 3, 4, 5, 6};
vector<int> v2 (v1.size());
copy(v1.begin(), v1.end(),
     v2.begin());
```

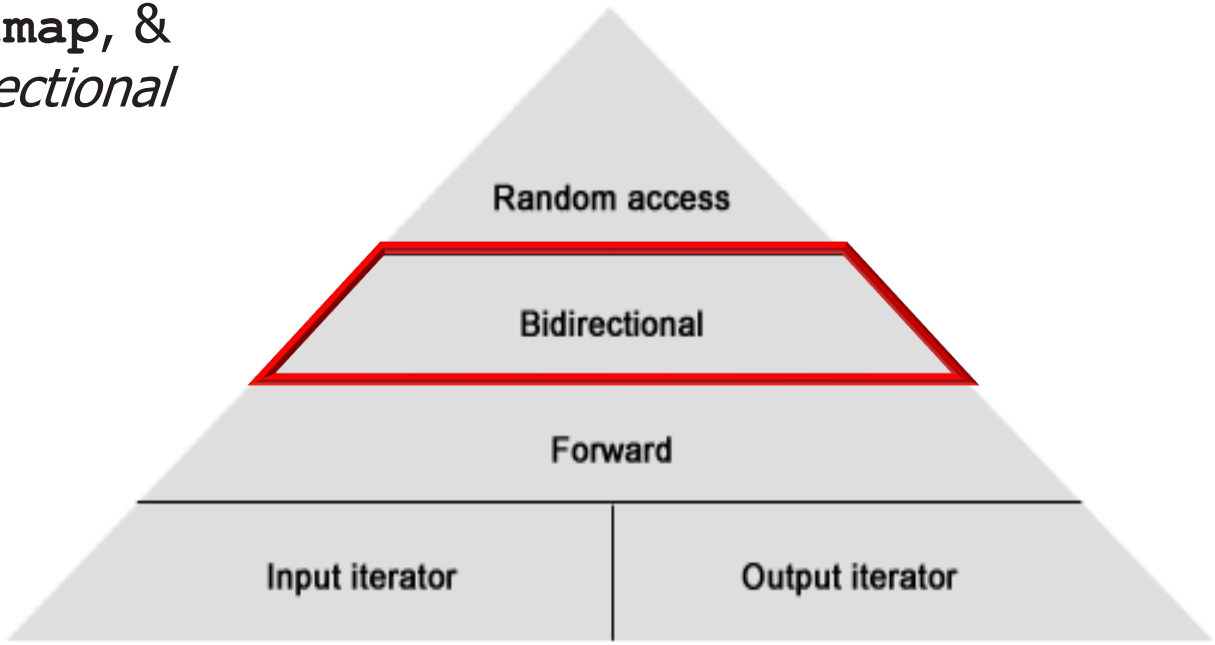
STL Generic Algorithms

- Each container declares an `iterator`, `const_iterator`, etc.as traits

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
    typedef T                                       value_type;
    typedef Allocator                               allocator_type;
    typedef typename allocator_type::reference      reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef __normal_iterator<pointer, vector>      iterator;
    typedef __normal_iterator<const_pointer, vector> const_iterator;
    typedef typename allocator_type::size_type      size_type;
    typedef typename allocator_type::difference_type difference_type;
    typedef typename allocator_type::pointer        pointer;
    typedef typename allocator_type::const_pointer  const_pointer;
    typedef std::reverse_iterator<iterator>         reverse_iterator;
    typedef std::reverse_iterator<const_iterator>   const_reverse_iterator;
```

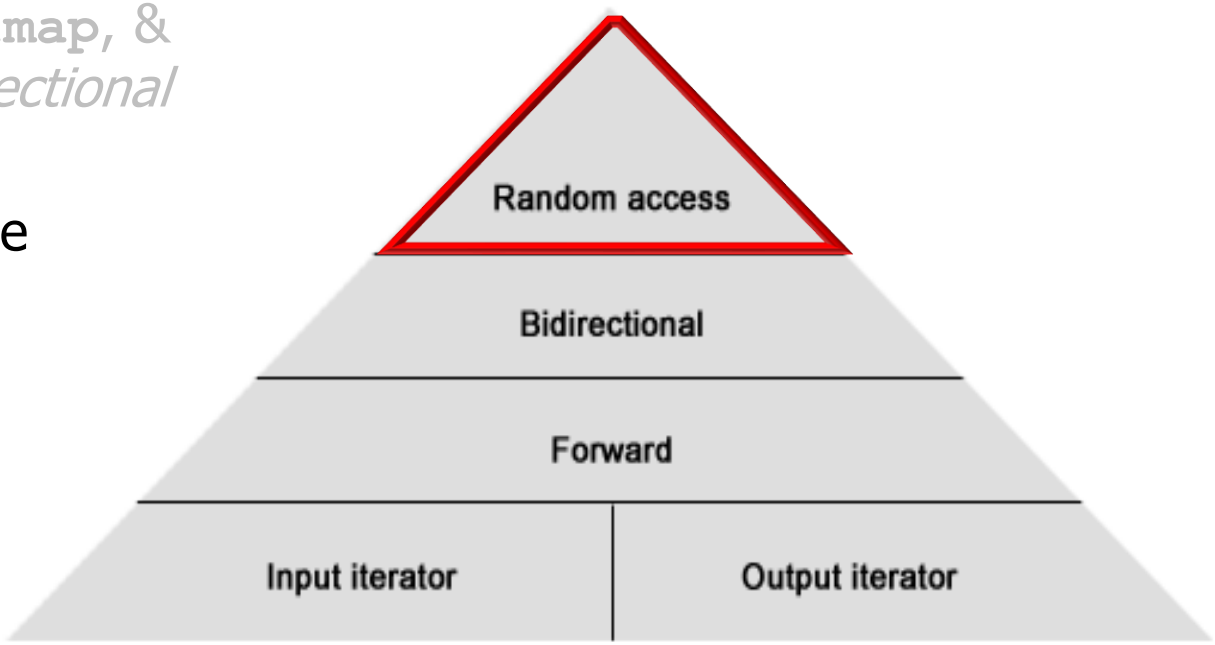
STL Generic Algorithms

- Each container declares an `iterator`, `const_iterator`, etc., as traits
  - `list`, `map`, `set`, `multimap`, & `multiset` declare *bidirectional* iterators



STL Generic Algorithms

- Each container declares an `iterator`, `const_iterator`, etc., as traits
  - `list`, `map`, `set`, `multimap`, & `multiset` declare *bidirectional* iterators
  - `vector` & `deque` declare *random-access* iterators



STL Generic Algorithms

- Each container declares factory methods for its iterator type

Iterators	
`begin()` `cbegin()`	returns an iterator to the beginning
`end()` `cend()`	returns an iterator to the end
`rbegin()` `crbegin()`	returns a reverse iterator to the beginning
`rend()` `crend()`	returns a reverse iterator to the end

STL Generic Algorithms

- Composing an algorithm with a container simply involves invoking the algorithm with iterators for that container

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

vector<int> v1 {1, 2, 3, 4, 5, 6};
vector<int> v2 (v1.size());
copy(v1.begin(), v1.end(),
     v2.begin());
```

STL Generic Algorithms

- Composing an algorithm with a container simply involves invoking the algorithm with iterators for that container
  - Templates provide compile-time type safety for containers, iterators, & algorithms


```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

vector<int> v1 {1, 2, 3, 4, 5, 6};
vector<int> v2 (v1.size());
copy(v1.begin(), v1.end(),
     v2.begin());
```

Categorizing STL Generic Algorithms

- There are four ways to categorize STL algorithms

Categorizing STL Generic Algorithms

- There are four ways to categorize STL algorithms, *e.g.*
  - **Non-mutating**, which operate using a range of iterators, but don't change the data elements found

See github.com/douglascraigschmidt/CPlusPlus/tree/master/STL/S-10

Categorizing STL Generic Algorithms

- There are four ways to categorize STL algorithms, *e.g.*
  - **Non-mutating**, which operate using a range of iterators, but don't change the data elements found
  - **Mutating**, which operate using a range of iterators, but can change the order of the data elements

See github.com/douglascraigschmidt/CPlusPlus/tree/master/STL/S-11

Categorizing STL Generic Algorithms

- There are four ways to categorize STL algorithms, *e.g.*
  - **Non-mutating**, which operate using a range of iterators, but don't change the data elements found
  - **Mutating**, which operate using a range of iterators, but can change the order of the data elements
  - **Sorting & sets**, which sort or searches ranges of elements & act on sorted ranges by testing values

See github.com/douglascraigschmidt/CPlusPlus/tree/master/STL/S-12

Categorizing STL Generic Algorithms

- There are four ways to categorize STL algorithms, *e.g.*
  - **Non-mutating**, which operate using a range of iterators, but don't change the data elements found
  - **Mutating**, which operate using a range of iterators, but can change the order of the data elements
  - **Sorting & sets**, which sort or searches ranges of elements & act on sorted ranges by testing values
  - **Numeric**, which are mutating algorithms that produce numeric results

Categorizing STL Generic Algorithms

- In addition to these main types, there are specific algorithms within each type that accept a predicate condition
  - Predicate names end with the `_if()` suffix to indicate they require an "if" test's result (true or false) as an argument

Categorizing STL Generic Algorithms

- In addition to these main types, there are specific algorithms within each type that accept a predicate condition
  - Predicate names end with the `_if()` suffix to indicate they require an "if" test's result (true or false) as an argument
    - Often used with functor calls

```
vector<int> v{1, 2, 3, 4, 5};
list<int> l{5, 4, 3, 2, 1};

auto vi =
  find_if(v.begin(),
          v.end(),
          bind(greater<>, _1, 5));

auto li =
  find_if(l.begin(),
          l.end(),
          not_fn(bind(greater<>,
                      _1, 5));
```

Benefits of STL Generic Algorithms

- STL algorithms are decoupled from the particular containers they operate on & are instead parameterized by iterators
  - All containers with the same iterator type can use the same algorithms

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

vector<int> v1 {1, 2, 3, 4, 5, 6};
vector<int> v2 (v1.size());
copy(v1.begin(), v1.end(),
     v2.begin());
```

Benefits of STL Generic Algorithms

- Since algorithms are written to work on iterators rather than components, the software development effort is drastically reduced

Benefits of STL Generic Algorithms

- Since algorithms are written to work on iterators rather than components, the software development effort is drastically reduced, e.g.,
  - Instead of writing a copy routine for each kind of container, one only write one for each iterator type & apply it any container

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

vector<int> v1 {1, 2, 3, 4, 5, 6};
list<int> l1 (v1.size());
copy(v1.begin(), v1.end(),
     l1.begin());
```

Benefits of STL Generic Algorithms

- Since algorithms are written to work on iterators rather than components, the software development effort is drastically reduced, e.g.,
  - Instead of writing a copy routine for each kind of container, one only write one for each iterator type & apply it any container
  - Since different components can be accessed by the same iterators, just one (or a few) version(s) of the copy routine must be implemented

```
template<typename InputIterator,
         typename OutputIterator>
OutputIterator copy
  (InputIterator first,
   InputIterator last,
   OutputIterator result) {
  for (; first != last;
       ++first, ++result)
    *result = *first;
    return result;
}

vector<int> v1 {1, 2, 3, 4, 5, 6};
vector<int> v2 (v1.size());
copy(v1.begin(), v1.end(),
     v2.begin());
```