STL Iterators

STL Iterators

- STL provides iterators for iterating over STL containers.
- Iterators are very powerful, as they let you iterate over any container.
- And, if you define your own container, and provide the iterator interface over that, you can use any of the STL Algorithms on your own container !

Iterators

- Just like templates make a class or function independent of the type, iterators help in making the STL algorithms independent of the container being used.

- This means that the iterator interface makes it such that the STL algorithms (as well as your own code) is independent of the internal implementation of the container class.

STL Iterators

- There are different types of iterators, based on the functionality that they expose:
  - Input iterators
  - Output iterators
  - Forward iterators
  - Bidirectional iterators
  - Random access iterators

Input Iterators

- These types of iterators are used for referring to some object, and they can be incremented to then refer to the next object in the sequence. They may not necessarily support multiple passes over the input.

- These are the most restrictive type of iterators, and pretty much all the STL iterators can behave as input iterators.
  - For example, vector<int>::iterator can behave as an input iterator, in addition to other types of iterators.
  - An example of just an input iterator type is the istream_iterator, and we will see a usage of this later in our code examples.

Output Iterators

- These iterators are used for storing values, but may not be usable for accessing them.
- Like input iterators, they may not support multiple passes.
- ostream_iterator is an example of such an iterator.

Forward Iterators

- A forward iterator goes only forward, i.e., you cannot decrement these or go back.
- However, you can restart the iterator and do multiple passes.
- An example is the iterators over the singly linked list class slist.
- slist<int>::iterator or slist<double>::iterator are forward iterators.
- Iterators over doubly linked list or vector are also forward iterators, but they are also bidirectional.
- In other words, bidirectional iterators are a super set of forward iterators.

Bidirectional Iterators

- As the name suggests, these iterators can go forward or backward.
- Iterators over list and vector and deque are examples of bidirectional iterators.
- These iterators can be decremented or incremented.

Random access Iterators

- These types of iterators do not need to move in steps of only 1.
- They can be incremented (or decremented) by any value N
  - as long as that increment(or decrement) keeps the iterator within a valid range over the container.
- Think of random access iterators as bidirectional iterators that have been enabled for random access.
- list iterators are not random access.
- vector and deque iterators are random access.

begin() and end()

- Member method begin() returns an iterator pointing to the beginning of the collection.
- Member method end() returns an iterator pointing to the end of the collection (actually one past the end, i.e., one past the last element)… Half open range.
- You should NOT dereference the iterator returned by a call to end().
- The iterator interface is same for all containers.
- This makes it such that iterators can be used in STL algorithms, and so the STL algorithms work on any container (even containers that are not part of STL, as long as they provide an iterator interface).

Example

```
std::vector <int> vi;
std::list< double > ld;
// ... Code to push data into the vector and list

 // citer1 and citer2 "point" to the first element of vi and ld respectively
std::vector<int>::const_iterator    citer1  =   vi.begin();
std::list<double>::const_iterator   citer2  =   ld.begin();

cout << *citer1 << " " << *citer2 << endl;  // prints the two elements

++ citer1;          // point to next element (2nd element) in vi
++ citer2;          // point to next element (2nd element) in ld
```

Example

```
// Make citer jump 5 elements, making it point to
// 7th element in vi (assuming there is a 7th element)
citer1 = citer1 + 5;

// Compilation error below, this kind of jump is not supported on lists.
// Why do u think that is the case ?
citer2 = citer2 + 5;

*citer1 = 5;        // compilation error, citer1 is a const iterator
*citer2 = 1.5;      // compilation error, citer2 is a const iterator
```

Example

```
// So, we need non-const iterators if we want to modify values
std::vector<int>::iterator   iter1  =  vi.begin();
std::list<double>::iterator  iter2  =  ld.begin();

*iter1 = 5;                  // changes first value to 5
*iter2 = 1.5;                // changes first value to 1.5


iter1 ++
*iter1 = 77;                 // changes second value to 77
```

Example

```
// Output values using citer1, can also use iter1 for this
citer1 = vi.begin();                    // reinitialize to start of vi


// vi.end() returns pointer to one past the last element
for ( ; citer1 != vi.end(); ++ citer )
{
        cout << *citer <<" " ;
}

cout << endl;
```

NOTE: In a later lesson, we will look at an alternative (more compact) way to output the values of a vector or list (or any STL collection, or more specifically, any collection that provides an iterator interface).

Example

- Iterators can be invalidated if you modify the collection (insert or delete).
- However, in the case of lists, the iterators are not invalidated, unless of course you remove the element that the iterator is pointing to.

rbegin and rend

- rbegin() and rend() are similar to begin() and end(), except they iterate from the end to beginning.

- Example:

```
vector< int > vi;
vector< int >::reverse_iterator  rIter = vi.rbegin();
for ( ;  rIter != vi.rend(); ++ rIter )
    cout << *rIter << endl;
```

STL functions on iterators

- advance
- void advance (iter, n);
  - This will move the iterator by the specified amount.
  - If this is an input iterator or forward iterator, this will increment the iterator n times. (n > 0)
  - If this is a bidirectional iterator, n can be positive or negative, and the iterator will be incremented (or decremented) n times.
  - If this is a random iterator, it will simply add n to the iterator (n can be positive or negative).

advance

```
template < class _InIt,  class _Diff> inline
void _Advance(_InIt& _Where, _Diff _Off, input_iterator_tag)
{           // increment iterator by offset, input iterators
            for ( ; 0 < _Off; --_Off)
                        ++_Where;
}

template <class _FwdIt, class _Diff> inline
void _Advance(_FwdIt& _Where, _Diff _Off, forward_iterator_tag)
{           // increment iterator by offset, forward iterators

            for (; 0 < _Off; --_Off)
                        ++_Where;
}
```

advance

```
template< class _BidIt, class _Diff> inline
void _Advance( _BidIt& _Where, _Diff _Off, bidirectional_iterator_tag)
{     // increment iterator by offset, bidirectional iterators

            for (; 0 < _Off; --_Off)
                        ++_Where;

            for (; _Off < 0; ++_Off)
                        --_Where;
}
```

advance

```
template<class _RanIt, class _Diff> inline
void _Advance(_RanIt& _Where, _Diff _Off, random_access_iterator_tag)
{           // increment iterator by offset, random-access iterators

            _Where += _Off;

}
```

advance complexity

- For random iterators, it is constant, because of pointer arithmetic.
- For others, it is linear O(N)

distance

- void distance(iterA, iterB);

- This function returns the distance (number of elements) between the two iterators (counting the first iterator)

- The second iterator should be reachable from the first iterator.

- For random iterator, it can just use subtraction to determine this.
- For other iterator types, it will increment iterA until it reaches iterB.

distance complexity

- For random iterators, distance has constant complexity, since it simply uses the operator –
- For other iterators, complexity is linear O (N)

back_inserter

- This is used to insert at the end of a container.
- The container should have push_back member method (deque, list, vector).

- Example:

```
void Foo()
{
        std::vector< double > vecD;
        for (int ii = 0; ii < 100; ++ ii)
                vecD.push_back( ii );

        std::list< double >  listD;
        std::copy (vecD.begin(), vecD.end(), back_inserter( listD) );
}
```

front_inserter

- This is used to insert at the front of a container.
- The container should have push_front member method (deque, list).

```
void Foo()
{
        std::deque< double > deqD;
        for (int ii = 0; ii < 100; ++ ii)
                deqD.push_back( ii );

        std::list< double >  listD;
        std::copy (deqD.begin(), deqD.end(), front_inserter( listD) );
}
```

inserter

- Creates an iterator for inserting elements at a given position in the container.
- The container should have insert member method.

```
void Foo()
{
        std::deque< double > deqD;
        for (int ii = 0; ii < 100; ++ ii)
                deqD.push_back( ii );

        std::list< double >  listD;
        std::copy (deqD.begin(), deqD.end(), front_inserter( listD) );
        std::list<double>::iterator  iter = listD.begin();
        advance( iter, 1 );

        std::copy( deqD.begin(), deqD.end(), inserter( listD, iter ) );

}
```