

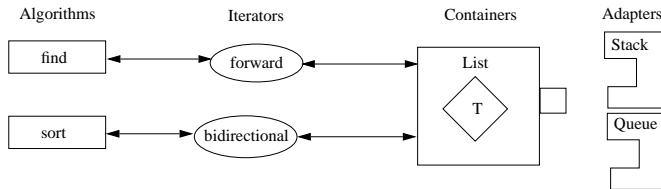
Iterators

There are different techniques for implementing iterators. So far, we have seen how you would implement a simple iterator template as a friend of a list template. All iterators contain state information about the first element of the list, the current element of the list, and can tell when the end of the list is reached.

The C++ Standard Template Library (STL) is based on four primary ideas:

1. Generic algorithms (e.g., find, sort, etc.)
2. Iterators on containers (e.g., forward iterator, bidirectional iterator)
3. Container types (e.g. the List template)
4. Adapters

Iterators are intermediary objects between a container and an algorithm that operates on that container.



Iterator Usage

We want to be able to use iterators just like we would use an integer or a pointer to iterate over an array, as in the following example:

```

List<int> list;
...
int sum = 0;
List<int>::iterator i;

for (i = list.begin(); i != list.end(); i++)
    sum += *i;
  
```

Consider the equivalent for just a simple array:

```

int list[n];
...
int sum = 0;
int* i;
int* begin = &list[0];
int* end = &list[n];

for (i = begin; i != end; i++)
    sum += *i;
  
```

The iterator abstraction provides the very much the same structure as iterating over an array using a pointer. The primary difference is that the iterator works with a linked list, and there is little change for error in pointer manipulation.

Containers and Iterators

A container should define the kind of iterator that is to be used with the container. For example, the List container requires a bi-directional iterator (which is both a forward and a backward iterator), which can traverse the list implementation in either direction.

A List container class could include a typedef for the kind of iterator that can use the List. You would also need to provide begin() and end() methods that returned iterator objects as results. Each iterator object contains information about a position in the list, and you can increment (operator++) and decrement (--) the iterator to move forward or backward through the list.

```

template <class T> class List {
private:
...
public:
...
    typedef BiIterator<T> iterator;
    friend class BiIterator<T>;

    iterator begin(); // return an iterator for the start of the list, or end if empty
    iterator end(); // return an iterator that is past the end of the list

    // insert new value after position defined by iterator, and return iterator to new node
    iterator insert(iterator position, const T& v);

    // remove the element at position
    iterator erase(iterator position);

    ...// insert_front, insert_back, etc.
};
  
```

Iterator Interface

The iterator interface must then provide overloaded versions of the operators that allow an iterator object to be treated as though it were a pointer, but in a safe manner.

```

template<class T> class BiIterator {
private:
    List<T>& list;

    friend class List<T>;

    BiIterator(List<T>& l); // can only be constructed by a List<T>

public:
    typedef BiIterator<T> iterator;
    ...

    bool operator == (const iterator& i) const;
    bool operator != (const iterator& i) const;

    iterator& operator++(); // pre-increment, ++i
    iterator& operator++(int); // post-increment ++i

    iterator& operator--(); // pre-decrement, --i
    iterator& operator--(int); // post-decrement, i--

    T& operator* () const;
};
  
```