CSCI-1200 Data Structures — Fall 2014
Lecture 9 — Iterators & Lists

Review from Lecture 8

- Explored a program to maintain a class enrollment list and an associated waiting list.
- Unfortunately, erasing items from the front or middle of vectors is inefficient.
- Iterators can be used to access elements of a vector

Today's Class

- Review of iterators and iterator operations
- STL `lists`, `erase` and `insert` on STL `lists`
- Differences between indices and iterators, differences between STL `list` and STL `vector`.
- Building *our own* basic linked lists:
  - Stepping through a list
  - Push back
  - ... & even more next week!

9.1 Review: Iterators and Iterator Operations

- An iterator type is defined by each STL container class. For example:

```
std::vector<double>::iterator v_itr;
std::list<std::string>::iterator l_itr;
std::string::iterator s_itr;
```

- An iterator is assigned to a specific location in a container. For example:

```
v_itr = vec.begin() + i;   //  i-th location in a vector
l_itr = lst.begin();       //  first entry in a list
s_itr = str.begin();       //  first char of a string
```

  Note: We can add an integer to vector and string iterators, but not to list iterators.

- The contents of the specific entry referred to by an iterator are accessed using the `*` *dereference operator*: In the first and third lines, `*v_itr` and `*l_itr` are l-values. In the second, `*s_itr` is an r-value.

```
*v_itr = 3.14;
cout << *s_itr << endl;
*l_itr = "Hello";
```

- Stepping through a container, either forward and backward, is done using increment (`++`) and decrement (`--`) operators:

```
++itr;   itr++;   --itr;   itr--;
```

  These operations move the iterator to the next and previous locations in the vector, list, or string. The operations do not change the contents of container!

- Finally, we can change the container that a specific iterator is attached to **as long as the types match**. Thus, if `v` and `w` are both `std::vector<double>`, then the code:

```
v_itr = v.begin();
*v_itr = 3.14;   // changes 1st entry in v
v_itr = w.begin() + 2;
*v_itr = 2.78;   // changes 3rd entry in w
```

  works fine because `v_itr` is a `std::vector<double>::iterator`, but if `a` is a `std::vector<std::string>` then

```
v_itr = a.begin();
```

  is a syntax error because of a type clash!

9.2 Additional Iterator Operations for Vector (& String) Iterators

- Initialization at a random spot in the vector:

  ```
  v_itr = v.begin() + i;
  ```

  Jumping around inside the vector through addition and subtraction of location counts:

  ```
  v_itr = v_itr + 5;
  ```

  moves `p` 5 locations further in the vector. These operations are constant time, $O(1)$ for vectors.

- These operations are *not allowed* for list iterators (and most other iterators, for that matter) because of the way the corresponding containers are built. These operations would be linear time, $O(n)$, for lists, where n is the number of slots jumped forward/backward. Thus, they are not provided by STL for lists.

- Students are often confused by the difference between iterators and indices for vectors. Consider the following declarations:

  ```
  std::vector<double> a(10, 2.5);
  std::vector<double>::iterator p = a.begin() + 5;
  unsigned int i=5;
  ```

- Iterator `p` refers to location 5 in vector `a`. The value stored there is directly accessed through the `*` operator:

  ```
  *p = 6.0;
  cout << *p << endl;
  ```

- The above code has **changed the contents** of vector `a`. Here's the equivalent code using subscripting:

  ```
  a[i] = 6.0;
  cout << a[i] << endl;
  ```

9.3 Implementing `Vec<T>` Iterators

- Let's add iterators to our `Vec<T>` class declaration from Lecture 7:

  ```
  public:
    // TYPEDEFS
    typedef T* iterator;
    typedef const T* const_iterator;

    // MODIFIERS
    iterator erase(iterator p);

    // ITERATOR OPERATIONS
    iterator begin() { return m_data; }
    const_iterator begin() const { return m_data; }
    iterator end() { return m_data + m_size; }
    const_iterator end() const { return m_data + m_size; }
  ```

- First, remember that `typedef` statements create custom, alternate names for existing types.

  `Vec<int>::iterator` is an iterator type defined by the `Vec<int>` class. It is just a `T *` (an `int *`). Thus, internal to the declarations and member functions, `T*` and `iterator` **may be used interchangeably**.

- Because the underlying implementation of `Vec` uses an array, and because pointers *are* the "iterator"s of arrays, the implementation of vector iterators is quite simple. *Note: the implementation of iterators for other STL containers is more involved!*

- Thus, `begin()` returns a pointer to the first slot in the `m_data` array. And `end()` returns a pointer to the "slot" just beyond the last legal element in the `m_data` array (as prescribed in the STL standard).

- Furthermore, dereferencing a `Vec<T>::iterator` (dereferencing a pointer to type `T`) correctly returns one of the objects in the `m_data`, an object with type `T`.

- And similarly, the `++`, `--`, `<`, `==`, `!=`, `>=`, etc. operators on pointers automatically apply to `Vec` iterators. We don't need to write any additional functions for iterators, since we get all of the necessary behavior from the underlying pointer implementation.

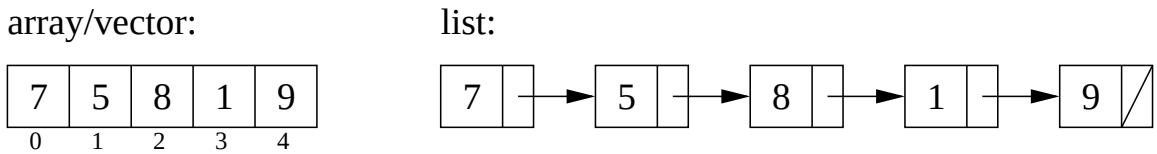
- The `erase` function requires a bit more attention. We've implemented the core of this function above. The STL standard further specifies that the return value of `erase` is an iterator pointing to the new location of the element just after the one that was deleted.

- Finally, note that after a `push_back` or `erase` or `resize` call some or all iterators referring to elements in that vector may be *invalidated*. Why? You must take care when designing your program logic to avoid invalid iterator bugs!

9.4 The `list` Standard Library Container Class

- Both lists & vectors store sequential data that can shrink or grow.

- However, the use of memory is fundamentally different. Vectors are formed as a single contiguous array-like block of memory. Lists are formed as a chain of separate memory blocks (not necessarily next to each other or in a particular order in memory), connected by pointers.



- STL `lists` have no subscripting operation (we can't use `[]` to access data). That is, they do not allow "random-access" or indexing. The only way to get to the middle of a list is to follow pointers one link at a time. Vectors do allow indexing (we can directly jump to the middle of the vector).

- STL `lists` have `push_front` and `pop_front` functions in addition to `push_back` and `pop_back`.

- `erase` and `insert` in the middle of the STL `list` is very efficient, independent of the size of the list. Both are implemented by rearranging pointers between the small blocks of memeory.

- Note: We can't use the standard `sort` function; we must use a special `sort` member function defined by the STL `list` type:

```
std::vector<int> my_vec;
std::list<int> my_lst;
// ... put some data in my_vec & my_lst
std::sort(my_vec.begin(),my_vec.end(),optional_compare_function);
my_lst.sort(optional_compare_function);
```

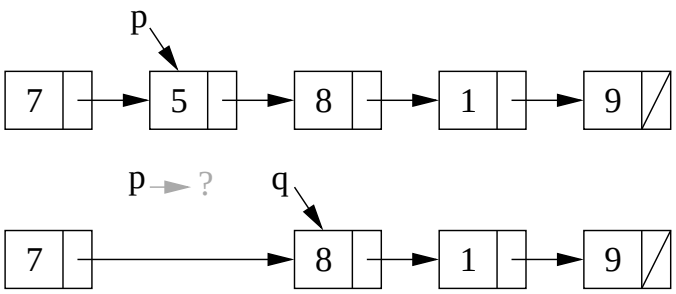
  Note: STL `list sort` member function is just as efficient, $O(n \log n)$, and will also take the same optional compare function as STL `vector`.

9.5 Erase & Iterators

- STL `lists` and `vectors` each have a special member function called `erase`. In particular, given list of ints `s`, consider the example:

```
std::list<int>::iterator p = s.begin();
++p;
std::list<int>::iterator q = s.erase(p);
```

- After the code above is executed:
  - The integer stored in the second entry of the list has been removed.
  - The size of the list has shrunk by one.
  - The iterator `p` does not refer to a valid entry.
  - The iterator `q` refers to the item that was the third entry and is now the second.



- To reuse the iterator `p` and make it a valid entry, you will often see the code written:

```
std::list<int>::iterator p = s.begin();
++p;
p = s.erase(p);
```

- Even though the `erase` function has the same syntax for vectors and for list, the vector version is $O(n)$, whereas the list version is $O(1)$.

9.6 Insert

- Similarly, there is an `insert` function for STL lists that takes an iterator and a value and adds a link in the chain with the new value immediately before the item pointed to by the iterator.
- The call returns an iterator that points to the newly added element. Variants on the basic insert function are also defined.

9.7 Exercise: Using STL `list` Erase & Insert

Write a function that takes an STL `list` of integers, `lst`, and an integer, `x`. The function should 1) remove all negative numbers from the list, 2) verify that the remaining elements in the list are sorted in increasing order, and 3) insert `x` into the list such that the order is maintained.

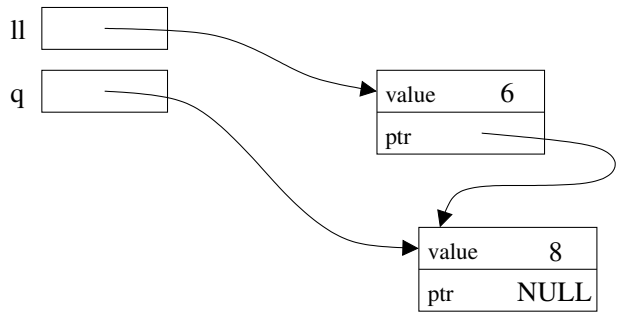
9.8 Working towards *our own* version of the STL `list`

- Our discussion of how the STL `list<T>` is implemented has been intuitive: it is a "chain" of objects.
- Now we will study the underlying mechanism — *linked lists*.
- This will allow us to build custom classes that mimic the STL `list` class, and add extensions and new features (more next week!).

9.9 Objects with Pointers, Linking Objects Together

- The two fundamental mechanisms of linked lists are:
  - creating objects with pointers as one of the member variables, and
  - making these pointers point to other objects of the same type.
- These mechanisms are illustrated in the following program:

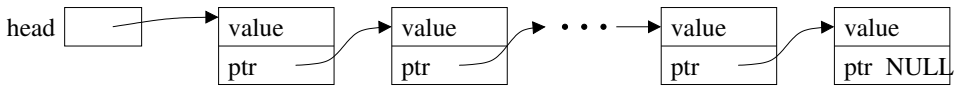
```
#include <iostream>

template <class T>
class Node {
public:
  T value;
  Node* ptr;
};

int main() {
  Node<int>* ll;       // ll is a pointer to a (non-existent) Node
  ll = new Node<int>;  // Create a Node and assign its memory address to ll
  ll->value = 6;       // This is the same as (*ll).value = 6;
  ll->ptr = NULL;      // NULL == 0, which indicates a "null" pointer

  Node<int>* q = new Node<int>;
  q->value = 8;
  q->ptr = NULL;

  // set ll's ptr member variable to
  // point to the same thing as variable q
  ll->ptr = q;

  cout << "1st value: " << ll->value << "\n"
       << "2nd value: " << ll->ptr->value << endl;
}
```



9.10 Definition: A Linked List

- The definition is recursive: A linked list is either:
  - Empty, or
  - Contains a node storing a value and a pointer to a linked list.
- The first node in the linked list is called the *head* node and the pointer to this node is called the *head* pointer. The pointer's value will be stored in a variable called `head`.

9.11 Visualizing Linked Lists



- The `head` pointer variable is drawn with its own box. It is an individual variable. It is important to have a separate pointer to the first node, since the "first" node may change.
- The objects (nodes) that have been dynamically allocated and stored in the linked lists are shown as boxes, with arrows drawn to represent pointers.
  - Note that this is a conceptual view only. The memory locations could be anywhere, and the actual values of the memory addresses aren't usually meaningful.
- The last node MUST have NULL for its pointer value — you will have all sorts of trouble if you don't ensure this!
- You should make a habit of drawing pictures of linked lists to figure out how to do the operations.

9.12 Basic Mechanisms: Stepping Through the List

- We'd like to write a function to determine if a particular value, stored in `x`, is also in the list.
- We can access the entire contents of the list, one step at a time, by starting just from the `head` pointer.
  - We will need a separate, local pointer variable to point to nodes in the list as we access them.
  - We will need a loop to step through the linked list (using the pointer variable) and a check on each value.

9.13 Exercise: Write `is_there`

```
template <class T> bool is_there(Node<T>* head, const T& x) {
```

9.14 Basic Mechanisms: Pushing on the Back

- Goal: place a new node at the end of the list.
- We must step to the end of the linked list, remembering the pointer to the last node.
  - This is an $O(n)$ operation and is a major drawback to the ordinary linked-list data structure we are discussing now. We will correct this drawback by creating a slightly more complicated linking structure in our next lecture.
- We must create a new node and attach it to the end.
- We must remember to update the `head` pointer variable's value if the linked list is initially empty.
  - Hence, in writing the function, we must pass the pointer variable **by reference**.

9.15 Exercise: Write `push_back`

```
template <class T> void push_back( Node<T>* & head, T const& value ) {
```