CSCI-1200 Data Structures — Fall 2022
Lecture 9 — Iterators Continued & Linked Lists

Review from Lecture 8

- Unfortunately, erasing items from the front or middle of vectors is inefficient.
- Introduction to iterators: for access, increment, decrement, erase, & insert
- Differences between indices and iterators
- STL's `list` class; differences between STL `list` and STL `vector`.

Today's Class

- *Being a user* of the STL `list` class & `list` iterators (Lab 5, Homework 4)
  - Big O Notation comparison of core `vector` & `list` operations
  - Syntax and functionality of insert & erase on STL `vector` & `list`
  - Situations for iterator invalidation
- *Finishing the implementation of our own version* of the `Vec` class
  - Implementation of iterators in our homemade `Vec` class (from Lecture 6)
- *Starting the implementation of our own* simple linked list:
  - Stepping through a list, searching for an element
  - Push front and push back
  - Insert in the middle
- BONUS: `const` and reference on return values

9.1 Compare & Contrast: STL `vector` vs. STL `list`

- Same: Both are templated, sequential *containers*.
- Different: Only `vector` can be accessed using subscript (a.k.a. random-access).
  (Note: Implementing a similar operation for `list` would be inefficient.)

```
std::vector<double> v(10, 3.14);
std::cout << v[4] << std::endl;
v[5] = 6.02;
```

- Same: Elements of both can be accessed by iterators, using the dereference operator. The syntax for iterators with `vector` and `list` was intentionally designed to be similar to pointers with arrays.

```
// std::vector<double> container(10, 3.14);
// std::vector<double>::iterator itr = container.begin();
std::list<double> container(10, 3.14);
std::list<double>::iterator itr = container.begin();
for (itr = container.begin(); itr != container.end(); itr++) {
  if (*itr < 0.0) {
    *itr = 0.0;
  }
}
```

- Same: Iterators can be incremented or decremented to visit all elements in order within the container.

```
++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!

- Same: We can use `==` and `!=` with vector, list, and string iterators.
- Different: We can use `<`, `<=`, `>`, and `>=` vector and string iterators, but not with list iterators. Why not?
  We'll talk about that in the next section...

- Different: Only `vector` iterators can jump forward (or backward) by an arbitary integer number of "slots". (Note: Implementing a similar operation for `list` would be inefficient.)

```
std::vector<double>::iterator v_itr = v.begin();
v_itr = v_itr + 5;
```

- Same: Both have `push_back` and `pop_back`. These operations are constant time for `list`, and constant time on *average* for `vector`.
- Different: Only `list` has `push_front` and `pop_front`. These are constant time, $O(1)$, operations. (Note: Implementing similar operations for `vector` would be inefficient.)
- Same: Both have `erase` and `insert`.

  Different: . . . however, while they are constant time, $O(1)$, operations for `list`, they are linear time, $O(n)$, operations for `vector`.
- Same: Both have a built in `sort` that runs in O(n log n), with an optional comparison function.

  Different: The syntax is slightly different.

```
std::sort(my_vector.begin(),my_vector.end(),optional_comparison_function);
my_list.sort(optional_comparison_function);
```

- Different: Situations in which iterators are *invalidated*.
  - Iterators positioned on an STL `vector`, at or after the point of an `erase` operation, are invalidated.
  - Iterators positioned anywhere on an STL `vector` *may be* invalid after an `insert` (or `push_back` or `resize`) operation. Why? Because the array might need to be resized & reallocated (re-located) on the heap.
  - Iterators attached to an STL `list` are not invalidated after an `insert` or `push_back`/`push_front` or `erase`/`pop_back`/`pop_front`. (Except iterators attached to the erased element!)

9.2 Implementing `Vec<T>` Iterators

- Let's add iterators to our `Vec<T>` class declaration from Lecture 6:

```
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! We'll see how STL* `list` *iterators work in a later lecture.*
- 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.

- Let's study the `erase` function. 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.

```
template <class T> typename Vec<T>::iterator Vec<T>::erase(iterator p) {
  for (iterator q = p; q+1 < m_data+m_size; ++q) {
    *q = *(q+1);
  }
  m_size --;
  return p;
}
```

9.3 Exercise: Write the `insert` function for our homemade `Vec` class.

Insert takes an iterator and a value as arguments, places the element immediately before the item pointed to by the iterator, and returns an iterator that points to the newly added element.
Unfortunately, just like erase, this function will have linear running time on average.

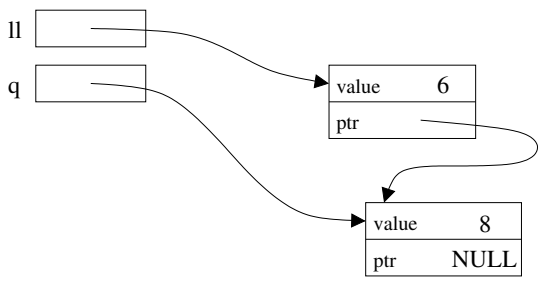
9.4 Working towards *our own* version of the STL `list`

- 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 in the next couple lectures!).

9.5 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:

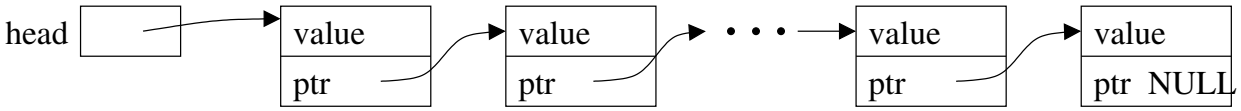
```
template <class T>
class Node {
public:
  T value;
  Node* ptr;
};

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;
}
```



9.6 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`.

9.7 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 variable 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.

9.8 Stepping Through the List, Searching for a Value

- We'd like to write a function to determine if a particular value, stored in `x`, is 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.

9.9 Exercise: Write `is_there`

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

- If the input linked list chain contains n elements, what is the Big O Notation of `is_there`?

9.10 Overview: Adding an Element at the Front of the List

- We must create a *new* node.
- We must permanently update the `head` pointer variable's value.
  Therefore, we must pass the pointer variable ***by reference*** .

9.11 Exercise: Write `push_front`

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

- If the input linked list chain contains n elements, what is the Big O Notation of the implementation of `push_front`?

9.12 Overview: Adding an Element at the Back 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 simple 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.

9.13 Exercise: Write `push_back`

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

- If the input linked list chain contains n elements, what is the Big O Notation of this implementation of `push_back`?

9.14 Inserting a Node into a Singly-Linked List

- With a singly-linked list, we'll need a pointer to the node *before* the spot where we wish to insert the new item. *NOTE: This is not how STL list's insert function works!*
- If `p` is a pointer to this node, and `x` holds the value to be inserted, write a *fragment of code* that will do the insertion. Draw a picture to illustrate what is happening.

- Note: This code will not work if you want to insert `x` in a new node at the *front* of the linked list. Why not?

9.15 Next time... Can we get better performance out of linked lists? Yes!

9.16 BONUS: References and Return Values

- A reference is an *alias* for another variable. For example:

```
string a = "Tommy";
string b = a;    // a new string is created using the string copy constructor
string& c = a;   // c is an alias/reference to the string object a
b[1] = 'i';
cout << a << " " << b << " " << c << endl;    // outputs:  Tommy Timmy Tommy
c[1] = 'a';
cout << a << " " << b << " " << c << endl;    // outputs:  Tammy Timmy Tammy
```

  The reference variable `c` refers to the same string as variable `a`. Therefore, when we change `c`, we change `a`.

- Exactly the same thing occurs with reference parameters to functions and the return values of functions. Let's look at the `Student` class from Lecture 3 again:

```
class Student {
public:
  const string& first_name() const { return first_name_; }
  const string& last_name() const { return last_name_; }
private:
  string first_name_;
  string last_name_;
};
```

- In the main function we had a vector of students:

```
vector<Student> students;
```

  Based on our discussion of references above and looking at the class declaration, what if we wrote the following. Would the code then be changing the internal contents of the i-th Student object?

```
string & fname = students[i].first_name();
fname[1] = 'i'
```

- The answer is NO! The `Student` class member function `first_name` returns a **const** reference. The compiler will complain that the above code is attempting to assign a const reference to a non-const reference variable.

- If we instead wrote the following, then compiler would complain that you are trying to change a const object.

```
const string & fname = students[i].first_name();
fname[1] = 'i'
```

- Hence in both cases the Student class would be "safe" from attempts at external modification.

- However, the author of the `Student` class would get into trouble if the member function return type was only a reference, and not a const reference. Then external users could access and change the internal contents of an object! This is a bad idea in most cases.