Announcement

- The coordination between lecture & lab material and between instructor & TAs was out-of-sync this week. I did not intend for iterators to be necessary for Lab 5. If you did not finish Lab 5 during your normal lab time, you are allowed to makeup ONE checkpoint from Lab 5 at the start of Lab 6.

Review from Lecture 9

- 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
- Iterators and iterator operations (increment, decrement, erase, & insert)
- STL's list class
- Differences between indices and iterators, differences between STL list and STL vector.

Today's Class

- Quick review of iterators – Questions about HW4?
- Implementation of iterators in our homemade Vec class (from Lecture 8)
- const and reference on return values
- Building our own basic linked lists:
  - Stepping through a list
  - Push back
  - ... & even more in the next couple lectures!

10.1 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!

10.2 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;
  ```
  
- Here's another common confusion:

  ```
  std::list<int> lst;
  lst.push_back(100);
  lst.push_back(200);
  lst.push_back(300);
  lst.push_back(400);
  lst.push_back(500);
  std::list<int>::iterator itr,itr2,itr3;
  itr = lst.begin(); // itr is pointing at the 100
  ++itr; // itr is now pointing at 200
  *itr += 1; // 200 becomes 201
  // itr ++ 1; // does not compile! you can't advance a list iterator like this
  ```

  ```
  itr = lst.end(); // itr is pointing "one past the last legal value" of lst
  itr--; // itr is now pointing at 500;
  itr2 = itr--; // itr is now pointing at 400, itr2 is still pointing at 500
  itr3 = --itr; // itr is now pointing at 300, itr3 is also pointing at 300
  ```

  // dangerous: decrementing the begin iterator is "undefined behavior"
  // (similarly, incrementing the end iterator is also undefined)
  // it may work for now, but it may break later on this machine or on another machine!

  ```
  itr = lst.begin();
  itr--; // dangerous!
  itr++;
  assert (*itr == 100); // might seem ok... but rewrite the code to avoid this!
  ```
10.3 Implementing Vec<T> Iterators

- Let's add iterators to our Vec<T> class declaration from Lecture 8:

```cpp
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 *. 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 array, 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 a version of this function in the previous lecture. 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!

10.4 References and Return Values

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

```cpp
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
```

- 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 4 again:

```cpp
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:

```cpp
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?

```cpp
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.

```cpp
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.

### 10.5 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!).

### 10.6 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:

```cpp
#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;
}
```
10.7 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.

10.8 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.

10.9 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.

10.10 Exercise: Write is_there

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

- If the input linked list chain contains n elements, what is the order notation of is_there?
10.11 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.

10.12 Exercise: Write push_front

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

- If the input linked list chain contains \( n \) elements, what is the order notation of the implementation of push_front?

10.13 Exercise: Write push_back

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

- If the input linked list chain contains \( n \) elements, what is the order notation of this implementation of push_back?

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