Lab 5 is due the same time as Lab 6

* "This" pointer – is a pointer to the implicit parameter.
  + List<int> a, b;
  + a.push\_back(10);
  + b.push\_back(11);
* List::push\_back(x)
  + Link<T> \* new\_link = new Link<T> (x) //everytime you do new you need to check
  + If(new\_link == null){
    - Cout << "Error";
    - Exit(1); }
  + //some kind of end case? What if the link list is empty
    - If(first\_link == null)
      * First\_link = new\_link;
      * Last\_link = new\_link;
  + New\_link -> prev\_link = last\_link; //puts the new links "prev link" equal to the old last link
  + Last\_link -> next\_link = new\_link;
  + Last\_link = new\_link;
  + My\_size++;

Iterating through the link list

For(Link<T> \*p = first\_link; p != null; p=p->next\_link)

Cout << p->value;

* List::erase(iterator(
  + Next = current\_link->next\_link;
  + Prev = current\_link->prev\_link;
  + Delete[] \*iter;
  + //possible end conditions, if there is no prev, if there is no next, and if the iter is pointing to the only link in the list
  + Remember to decrement size;
* List::insert(iterator)
  + Create a new link
  + Have the new link's prev link point to the original prev link
  + Then have the new links next point to the iterator link
  + Then have the prev link point to the new link,
  + Then have the next link point back to the new link
  + End conditions, if the list is empty then use pushback
* Pop back would use the erase function to erase the last link
* Template <class T>
* Void List<T>::pop\_back()
* {
  + Iterator temp(last\_link);
  + Erase(temp); // or this->erase(temp);
* }
* Template <class T>
* List\_iterator<T> &List\_iterator<T>::operator++() //preincrement
  + Current\_link = current\_link->next\_link;
  + Return \*this;
* Template <class T>
* List\_iterator<T> List\_iterator<T>::operator(int) //post increment
  + List\_iterator<T> copy(\*this);
  + Current\_link = current\_link -> next\_link; //or current\_link = ++\*this;
  + Return copy;