As with previous chapters, we are going to lay out the learning objectives for this chapter, and split them into what you should get on the first reading vs the second reading.

1 First Reading: Understand Key Ideas

On your first reading, the goal is to get the big ideas, and build the mental scaffolding in which to hold the details on your second reading—the learning objectives for your first reading are generally **Remember** and **Understand** objectives. If you are missing a few details (but not completely lost), it is ok to move ahead, and come back to them the next time through—you might ask questions on the course forums too! Note you should **sleep at least one night between the first and second reading** to give your brain time to process the information.

- **LinkedLists: Intro**
  - **Define** *linked list.*
  - **Explain** why we need a Node class and why it is good for it to be a private inner class of LinkedList.
  - **Explain** how the Node class is a recursive data type.
  - **Define** *head pointer.*
  - **Explain** how elements are stored in a LinkedList (in terms of the head pointer and next pointers).
  - **Define** *singly linked list.*
  - **Define** *doubly linked list.*
  - **Define** *tail pointer.*
  - **Define** *circular list.*
- **Linked List Basic Operations**
  - **Explain** how to add to the front of a singly or doubly linked list.
  - **Explain** how to add to the back of a singly or doubly linked list.
  - **Explain** how to add to search a linked list for a particular element.
  - **Explain** how to implement the rule of three methods for a linked list.
  - **Explain** the complexity/difficulty in adding in sorted order (or any other addition where you need to find a particular place) using a single pointer to a node.
  - **Explain** how to add to linked list in a particular place using a `Node *` and iteration.
  - **Explain** how to use recursion to add to a linked list in a particular place.
  - **Explain** how to add to a linked list in a particular place using a `Node **`.
    - Note: if you are taking ECE 550, you should have learned that one of the key rules of computing is "If you have a functionality problem, add a level of indirection"—this approach is an instance of that rule.
  - **Explain** how to remove from the front (or back if doubly linked with a tail pointer) of a linked list.

- **Explain** how to remove a particular element from a linked list using a `Node *` and iteration.
- **Explain** how to remove a particular element from a linked list using recursion.
- **Explain** how to remove a particular element from a linked list using a `Node **` and iteration.
- **Recognize** that fields of an object should reflect the state of the object, and not be "hacks" to store extra information for badly designed code.
- **Explain** why removing all elements by repeatedly removing one element is inefficient.
- **Explain** how we removed the duplication of code in a good way for removeAll.
  - Note: some students wonder why we used a template with a boolean parameter rather than passing a boolean as a regular function parameter. Because the template parameter is guaranteed to be a compile-time constant and the compiler will create a specialization with each of the values of removeAll, it is easier for the compiler to optimize out the `if` statement. Compilers might optimize it out even with the regular function parameter, but as Drew cares a lot about performance, writing it in the easiest to optimize way is the most natural for him.

- **Iterators**
  - **Recall** *iterators* (from Chapter 17).
  - **Explain** why iterators are important for access to elements of a linked list from an efficiency standpoint.
  - **Explain** why iterator is a *public inner class* of linked list.
  - **Explain** what state a linked lists's iterator class needs to hold and why.
  - **Explain** the general idea of how to implement operations on a linked list's iterator class.

- **Uses for ADTs**
  - **Explain** the high-level advantage of linked lists compare to arrays for implementing several ADTs.
  - **Explain** how to implement a stack with a linked list.
  - **Explain** how to implement a queue with a linked list.
  - **Explain** how to implement a set with a linked list.
  - **Explain** how to implement a map with a linked list.

- **STL List**
  - **State** the name of the STL class which implementes a doubly linked list.

2 Second Reading

As always, we strongly recommend you sleep between your first and second reading. The second reading is where you want to focus on the higher-level learning objectives, which build on the lower-level learning objectives you worked on in the first reading. Sleeping gives your brain a chance to process the information from the first reading.

- **Draw** a diagram of a (singly or doubly) linked list with a particular set of elements, and transform the diagram to show how various operations are performed on it. Note that you should be able to do this learning objective at a couple different levels of abstraction:
  - When working through the code within a linked list to show (and maybe debug) what the operation is doing.
  - When working through code that uses a list, drawing the net result of the operation.
- **Write** code to implement linked list operations (not only those described here, but others), as well as iterators within linked lists.
  - Note: there are plenty of interview questions to do something (possibly weird) to a linked list. Google "linked list interview questions" and you'll find more than you can count.
- **Write** code that uses a linked list (including iterators within the list). Note: this will mostly be with `std::list` rather than your own list.
- **Determine** the time complexity of code with linked lists.
- **Use** the idea of pointer-to-a-pointer-to-X in other situations where it can help make code cleaner (Note: this technique can be great in a lot of situations—we'll see it again in the next chapter, but we want you to start having it in your toolkit for general purpose use).
- **Assess** if a linked list is an appropriate data structure for what you need to do.
- A longer-term objective: **design** other linked structures based on the idea of linked lists when they are appropriate to solve your problem. You don't really need to do this now, but its a good longer term goal. We'll also note that "blockchain" is basically a linked list with special features.

3 Key Learning Objectives

We want you to "get good" with linked lists in three ways:

1. Drawing diagrams of the program state and how it changes (as discussed above, at two different levels of abstraction)
2. Implementing a linked list, including possibly unusual operations (especially for interviewing)
3. Using a linked list in your code, when appropriate.

We'll note that beyond being good at linked lists for their own use, you will have an easier time with the next chapter if you are good at linked lists.