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.

- **Intro**
  - **List** the two facets of high-performance programming.
- **Big-Oh Notation**
  - **Define** *asymptotic behavior.*
  - **Define** *constant factors.*
  - **Recognize** that there is a formal mathematical definition for Big-Oh, as well as a limit test for it. Note that the specific definition and applying the limit rule are not learning objectives of this course: you would probably go much deeper into the math in a theory course.
  - **Explain** the characteristics of a constant time algorithm, and **state** its Big-Oh.
  - **Explain** the characteristics of a log star time algorithm, and **state** its Big-Oh.
  - **Explain** the characteristics of a logarithmic time algorithm, and **state** its Big-Oh.
  - **Explain** the characteristics of a linear time algorithm, and **state** its Big-Oh.
  - **Explain** the characteristics of a quadratic time algorithm, and **state** its Big-Oh.
  - **Explain** the characteristics of a cubic time algorithm, and **state** its Big-Oh.
  - **Explain** the characteristics of a polynomial time algorithm, and **state** its Big-Oh.
  - **Define** *tractable.*
  - **Explain** the characteristics of a exponential time algorithm, and **state** its Big-Oh.
  - **Define** *intractable.*
  - Note: we don't actually define NP-complete problem, as that definition requires concepts (namely Turing Machines) that we don't introduce. A theory class should give you a definition of this important class of problems.
  - **Recognize** that the *best known* algorithms for NP-complete problems are exponential time (and we don't know if we can do better, but most people think we can't).
  - **Explain** the characteristics of a factorial time algorithm, and **state** its Big-Oh.
  - **Define** *amortized behavior.*
  - **Explain** how/why Big-Oh can be applied to an algorithm's space complexity.

  - **Explain** why Big-Oh is not a perfect measure of algorithm efficiency
    - **Note:** Many students get an unhealthy obsession with Big-Oh. Realizing how/when it is useful and how/when it is not is key.

- **Abstract Data Types**
  - **Define** *abstract data type (ADT).*
  - **Explain** the relationship between ADTs and abstraction.
- **Queues**
  - **Define** *queue.*
  - **Define** *FIFO.*
  - **Explain** some uses of queues.
  - **Explain** what the common operations in a queue ADT do.
  - **Name** the STL class which implements a queue.
  - **Explain** how to implement a queue with an array (or vector).
  - **Define** *deque.*
  - **Name** the STL class which implements a deque.
- **Stacks**
  - **Define** *stack.*
  - **Define** *LIFO.*
  - **Explain** some uses of stacks.
  - **Explain** what the common operations in a stack ADT do.
  - **Name** the STL class which implements a stack.
  - **Explain** how to implement a stack with an array (or vector).
- **Sets**
  - **Define** *set.*
  - **Define** *multiset* (or *bag*).
  - **Explain** some uses of sets.
  - **Explain** what the common operations in a set ADT do.
  - **Name** the STL class which implements a set.
  - **Explain** how to implement a set with an array (or vector).
- **Maps**
  - **Define** *map.*
  - **Explain** some uses of maps.
  - **Explain** what the common operations in a map ADT do.

  - **Name** the STL class which implements a map.
  - **Explain** how to implement a map with an array (or vector).
    - Note: you actually did this in C without knowing it was a map in the `kvs` assignments at the end of the C portion of this course.

- **ADTs and Abstract Classes**
  - **Explain** why abstract classes are appealing for ADTs from a software engineering perspective.
  - **List** and **explain** two problems with using abstract classes for ADTs in C++.
  - **Recognize** that there is a way to solve the iterator issue (as shown in AoP). Note that the details of this are not a learning objective of this class, but the general technique is good to understand in your C++ programming beyond this course.

2 Second Reading

As always, we strongly recommend you sleep between your first and second reading. However, this chapter is rather different than most. We need to introduce just enough theory (in terms of Big-Oh) to understand why we want some data structures you will learn about in the upcoming chapters. However, this class is not a theory course, so we aren't going deep into the theory. To that end, we want you to be able to **determine** the time complexity (Big-Oh) of algorithms, but only in cases where we don't need more theory than presented here.

We also want you to understand the distinction between an ADT (the *interface*) and the data structure (the *implementation*). In particular, we want you to understand that many data structures can be used to implement a variety of different ADTs, and that many ADTs can be implemented with a variety of different data structures.

For all the ADTs we discussed, we talked about how to implement them with an array (or vector), and you should already have the skills to **write** code to do so.

We want you to start working on the skill of **select** an appropriate ADT (or combination of them, maybe with some non-standard operations added) to solve a programming problem. Over the next several chapters, you will work on **select** an appropriate data structure (or combination of them) to *efficiently* implement the ADTs/operations you need.

We still recommend a second reading of this chapter, but note that it has a lot more "factual knowledge" to lay the foundation for the upcoming chapters.