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.

- **Templates: intro**
  - **Define** *polymorphism*.
  - **Define** *parametric polymorphism*.
  - **Define** *type parameter*.
  - **State** the C++ construct which provides parametric polymorphism.
- **Templated Functions**
  - **State** the syntax for declaring a templated function.
    - Note: you may see `template<class T>` instead of `template<typename T>`. There is NO difference. We strongly prefer typename for readability: it makes it clear that you can use any type there. You can still use any type with class (there is no difference), but someone might think you can only use a class type.
  - **Define** *instantiate* (a templated function).
  - **State** the syntax for instantiating a templated function.
  - **Define** *template specialization*.
  - **Explain** how templated functions solve the problem of duplicating code (*e.g.*, arrayMax at the start of the chapter).
- **Templated Classes**
  - **State** the syntax for declaring a templated class.
  - **State** the scope of the template parameters in a templated class.
  - **State** the syntax for instantiating a templated class.
  - **Recognize** that a templated class instantiated on different parameters are distinct types (*e.g.*, `MyTemplate<T1>` and `MyTemplate<T2>` are different types).
  - Note: Templates as Template Parameters is for completeness and won't come up in this class (unless you are taking 751).
- **Template Rules**

- **Explain** the rule "Template Definitions Must Be Visible at Instantiations", including *why* this rule exists, and *what* you need to do when writing templated classes for your code to compile properly.
- **Define** *compile-time constant*.
- **Explain** the rule "Template Arguments Must Be Compile-Time Constants", including *why* this rule exists, and *what* you need to do when writing templated classes for your code to compile properly.
- **State** when a template is type checked.
- **Explain** what benefit you gain (that is, what you can do that you could not otherwise do) by the fact that templates are only type checked when a specialization is created.
- **Explain** what benefit you gain by the fact that templated classes are only type checked on what is needed to create the object layout, and then on methods that are *actually* used.
- **Explain** the rule "Multiple Close Brackets Must Be Separated by Whitespace".
- **Explain** why you need to know if an identifier names a type or a value to determine what certain expressions mean (*e.g.*, you need to know if `x` names a type or a value to determine what `x * y` means).
- **Explain** why the above requirement is not a problem for C, or for C++ code without templates.
- **Recall** what the scope resolution operator (`::`) means, and in particular what `T::t` means (from Chapter 14).
- **Define** *dependent name*.
- **Explain** the rule "Dependent Type Names Require Keyword `template`", including *why* this rule exists, and *what* you need to do when writing templated classes for your code to compile properly.
- Note: We mention explicit specializations for completeness, but they are not a focus of the course, and won't come up in any assignments (unless you are doing 751).
- **Explain** what the rule "Template parameters for Functions (But Not Classes) May Be Inferred" means.

- **The Standard Template Library**
  - **Explain** what `std::vector<typename T>` is used for, and what the type parameter `T` represents.
  - Note: we mention a lot of methods and operators in vector. You are going to want to familiarize yourself with them and make use of them. You'll learn the most common ones the most quickly and others over more time. As always, do NOT memorize. Do know where to go look for more info!
  - **Recall** *lexicographic ordering* (from Chapter 10).
  - **Explain** how the `std::pair<typename T1, typename T2>` class is used.
  - **Explain** why the `std::pair<typename T1, typename T2>` class is useful.

- **Define** *container class*.
- **Explain** why `someThing[i]` may be inefficient, depending on what `someThing` is.
- **Define** *iterator*.
- **Explain** how iterators resolve the tension between efficiency and abstraction.
- **Explain** the type `std::vector<int>::iterator`
- **State** what `.begin()` and `.end()` do on `std::vector` (and many other STL classes)
- **State** what the `*` and `++` operators do on iterators.
- **Explain** why we prefer the prefix `++` operator for object types.
- **Explain** how `.begin()` is overloaded to return an `iterator` or a `const_iterator` appropriately.
- **Define** *invalidate* (an iterator).
- **Explain** how you know if a particular operation invalidates a particular iterator.
- **Define** *functional call operator*.
- **Explain** how to write a class with an overloaded function call operator to create a custom ordering (*e.g.*, for std::min or std::max).

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.

Before we dive into the learning objectives for the second reading, we want to note that they fall broadly into two categories: the mechanics of templates and using STL. For the former, you should be working towards a fairly solid mastery of these learning objectives in this chapter (both your second reading and the activities you do in class). For the second—using STL—you should get started towards these goals, but know that it will take you a long time to get familiar with STL. Vector is a great place to start (and you should start using it instead of C-style arrays!) but even that is a bit of a task. Vector is a big class (it has about 30 methods in it—plus some constructors, and inner classes like iterator). We wouldn't expect you to absorb all that at once. We do, however, want you to know to go to `https://cplusplus.com/reference/vector/vector/` to look things up as needed.

We'll also note that sometimes student ask/complain that we should "teach more of STL in this Chapter." There are two reasons why we don't. First, a lot of STL classes won't make sense until we get to the later Chapters of this book. Second, even just `vector` and `pair` have a lot for you to learn before you move on to other things.

- **Templates**
  - **Determine** when/how code duplication could be reduced by use of templates.
    - Note: this starts with a "now" component ("I have these two functions, they do the same thing, and could be templated") and over time will build into an "in the future" component ("I am writing this function, but I suspect I may want to use it for many types, so I will template it now in anticipation of my future needs").

- **Show** the specialization that is created for a particular template (function or class) instantiation[1].
  - Note: if you can do the above, there is nothing special for you to do to **execute** code with templates by hand.
- **Write** templated functions.
- **Write** templated classes.
- **Determine** whether an identifier is a dependent name in template code.
- **Use** `typename` properly for dependent type names.

- **STL**
  - **Execute** code by hand that has `std::vector` and `std::pair`. (Adding other STL classes later, either explicitly covered in this book, or by looking them up).
  - **Write** code with `std::vector` and `std::pair` (Note: stop using C-style arrays).
  - **Execute** code by hand that has iterators.
  - **Write** iterators inside classes you create. Note: we won't practice this LO until Ch 21, but you should start thinking about it.
  - **Write** code that uses iterators that are already defined in another class.
  - **Select** algorithms from STL where appropriate, and **write** code that makes use of them.

3 Key Learning Objectives

As noted above, the key learning objectives are to **execute** code with templates by hand and **write** your own code with templates (which may involve **determining** that a template will help reduce code duplication).

We want you to *start* working on the LOs related to STL—especially vector. However, this takes time. Becoming fluent in STL will help you write big complex C++ programs efficiently.

[1] Wow, you need like 3 of the learning objectives from the first reading just to make sense of this learning objective!