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.

- **Inheritance: Intro**
  - **Define** *inheritance*
  - **Define** *child class* (also *subclass*, *derived class*)
  - **Define** *parent class* (also *superclass*, *base class*)
  - **Define** *is-A relationship.*
  - **Define** *extends* (in the context of inheritance).
  - **Explain** why inheritance is useful.
  - **Define** *has-A relationship.*
  - **Define** *composition*
- **Another Conceptual Example**
  - **Define** *inheritance hierarchy.*
  - **Define** *override* (a method).
  - **Identify** what fields and methods exist in a child class (notably, what fields and methods are inherited)
  - **Explain** the relationship of classes in an inheritance hierarchy.
- **Writing Classes With Inheritance**
  - **State** the syntax to declare a class which inherits from another class.
  - **Recognize** that you should not re-declare fields in the child class which are inherited from the parent class.
  - Note: For completeness, AoP describes how to access fields if you declare a field in the child class with the same name as the parent. Don't worry about that so much: just don't re-declare fields.
  - Note: For completeness, AoP mentions inheriting from a parent class with private or protected—this pretty much never comes up: just always use public inheritance (class Child : public Parent).

- **Explain** what code can access a field or method declared `protected` (Note: the "slightly subtle restriction" is one of those things that you mostly need to know so you can understand what is going wrong if you break the rule).

- **Construction and Destruction**
  - **Explain** how objects of classes which use inheritance are constructed and destructed.
  - **Explain** how the type of an object in C++ changes during construction and destruction.
  - **Recognize** that the behavior of types changing during construction/destruction is fairly C++-specific: most other OO languages do it differently.
  - **State** the syntax for passing parameters to the parent class's constructor.

- **Subtype Polymorphism**
  - **Recall** the definition of *polymorphism* (from Chapter 17).
  - **Recall** the definition of *parametric polymorphism* (from Chapter 17).
  - **Define** *subtype*.
  - **Define** *subtype polymorphism*.
  - **Explain** why in C++ (and most other OO languages) a child class is a subtype of its parent class.
  - **Recognize** that in C++ subtype polymorphism only works with pointers and references
  - **Explain** why subtype polymorphism is useful (both in terms of reducing code duplication and in terms of future flexibility).
  - **Define** *static type*.
  - **Define** *dynamic type*.
  - **Explain** how the static type and dynamic type of an expression might differ.
  - **Explain** the practical restrictions of the fact that the compiler only works with static types.

- **Method Overriding**
  - **Explain** why method overriding is useful.
  - **Define** *static dispatch*.
  - **Explain** why static dispatch is not typically what we want for overridden methods.
  - **Define** *dynamic dispatch*.
  - **State** the syntax for making a method be dynamically dispatched, and *where* this declaration must be made.
  - **Explain** how to determine what method is called in C++ when the static and dynamic types of the object the method is called on differ.
  - **Recognize** that C++'s choice of static dispatch as the default is not common in other OO languages.
  - **Explain** how we will draw objects that have virtual methods, and why that is useful as you execute code by hand.

- **Explain** why it is important for classes that participate in subtype polymorphism to have virtual destructors.
- **State** the syntax to explicitly call the method of the parent class.
- **State** the in C++ with regards to how overridden methods may have (and may not) have different access modifiers than the method they are overriding.
- **Define** *covariant*.
- **Explain** why the return type of an overridden method may be different from that of the method it overrides, as long as it is covariant.

- **Abstract Methods and Classes**
  - **Define** *abstract method* (also: *pure virtual function*).
  - **State** the syntax for declaring a method as abstract.
  - **Define** *abstract class*.
  - **Define** *concrete subclass*.
  - **Explain** what you can do with an abstract class.
  - **State** the rule that abstract classes may not be instantiated.
  - **State** the rule that concrete subclasses of an abstract class must provide implementations for all abstract methods.
  - **Explain** how the two rules above work together to ensure that all method calls have an actual implementation.
  - **Explain** the "hole" in the above guarantee, why it exists in C++, and why it does not exist in Java.

- **Inheritance and Templates**
  - Note: This section is primarily to help you avoid trouble when writing your own classes—to make sure you know what you can and can't do to help avoid confusion and surprise when you are writing code.
  - **Recall** *composability* (from Chapter 4)
  - **List** three ways that templates and inheritance are naturally composable.
  - **Explain** the rule "A templated method cannot be virtual" and **explain** how you can work around this restriction if you only need a few known types.
  - **Explain** the rule "A templated function cannot override an inherited method" and what happens if you try it.
  - **Explain** the rule "Virtual methods are specialized when an instance is made" and what this rule means for type checking template classes that use virtual methods.

- **Planning Your Inheritance Hierarchy**
  - Note: Students sometimes express frustration that we don't go deeper into OO design here. This section barely scratches the surface of OO design. Details are left to a Software Engineering class (ECE 651 spends about a third of the semester on OO design).

  - **Explain** the process for planning your inheritance hierarchy.
    * Note: most of this section is just an example to help you understand the process outlined at the start.
  - **Recognize** that dynamic dispatch is preferred over conditional decisions on the type of object you have.
  - **Recognize** that a UML class diagram depicts the relationship between classes.
  - **Recognize** that there are a lot more topics on OO-design, and this was just the start of how to work on a class hierarchy.

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.

- **Categorize** the relationship between two types as "is-A", "has-A", or neither (Note: "both" is also valid, but won't come up in this class).
- **Execute** code by hand that involves inheritance, including: construction and destruction of objects, tracking the type of objects in your diagram, determining whether a method call is dispatched statically or dynamically, and correctly performing the (static or dynamic) method dispatch.
- **Compare and Constrast** *parametric* polymorphism and *subtype* polymorphism.
- **Assess** when subtype polymorphism is an appropriate technique for reducing code duplication.
- **Assess** when it is appropriate for a subclass to override a method from the parent class.
- **Assess** when it is appropriate for a method in a class to be abstract (aka "pure virtual").
- **Assess** when it is appropriate for a class to be abstract.
- **Write** code which makes use of inheritance, overridden methods, subtype polymorphism, dynamic dispatch, and/or abstract methods and classes.
- **Determine** when a set of classes/methods is likely to have significant code duplication.
- **Recommend** inheritance relationships between classes (*e.g.*, introducing a common parent).
- **Design** a class hierarchy, making good use of inheritance and composition.
- Note: as we previously mentioned there is a lot more to OO design than this chapter covers. We only expect you to do these last two learning objectives within the ideas presented here (and in previous chapters), not the more complete coverage you would get in Software Engineering. Your OO design skills will start here, and grow over the long term.

3 Key Learning Objectives

There are two main areas we want you to work to a solid mastery of in this chapter:

- **Execute** code by hand that involves inheritance, including: construction and destruction of objects, tracking the type of objects in your diagram, determining whether a method call is dispatched statically or dynamically, and correctly performing the (static or dynamic) method dispatch.
- **Write** code which makes use of inheritance, overridden methods, subtype polymorphism, dynamic dispatch, and/or abstract methods and classes.

One thing to start on with a much longer time-horizon for mastery (especially as we only scratched the surface here):

- **Design** a class hierarchy, making good use of inheritance and composition.

This learning objective is supported by many of the other learning objectives from the second reading. It also has a strong relationship with the LO to **write** code—a well-design inheritance hierarchy will make that code writing go better.