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.

- **Writing Code: Intro**
  - **Explain** why planning is important to programming
  - **Explain** differences between novices and experts in planning.
- **Step 1: Work an Example Yourself**
  - **Explain** how to do Step 1 of The Seven Steps.
  - **Explain** what considerations are important (and why) in choosing the specific instance to work for Step 1.
  - **Name** two techniques that you can use in Step 1 (which we used in this example) to help you with the algorithmic thinking process.
  - **Explain** how these two techniques can help you.
  - **Explain** how you might deal with the answer being "too obvious".
- **Step 2: Write Down What You Just Did**
  - **Explain** how to do Step 2 of The Seven Steps.
  - **Explain** how the Everything is a Number principle relates to Step 2.
  - **Explain** what the steps your write down for Step 2 should look like in general.
- **Step 3: Generalize Your Steps**
  - **Explain** the idea of "generalizing values".
  - **Explain** what to do if you are stuck on Step 3.
  - **Define** *mis-generalization.*
  - **Explain** where we are likely to discover mis-generalizations and what to do about them?
  - **Explain** the idea of "generalizing repetitions".
  - **Recognize** repetitive steps, which differ only in counting.
  - **Define** *identity element.*
  - **Explain** how the identity element can help make steps have a repetitive pattern.

  - **Explain** why the identity element is generally the correct answer when performing operations on an empty list.
  - **Explain** the idea of "generalizing conditional behavior".

- **Step 4: Test Your Steps**
  - **Explain** why you should test your algorithm on *different* inputs than you used in the first three steps.
  - **Explain** some guidelines for how to choose good cases to test with in Step 4.

- **Step 5: Translate Your Algorithm To Code**
  - **Define** *comments*.
  - **State** the syntax for each of the two types of comments in C.
  - **Explain** how to make use of your algorithm from Step 3 as a starting point for writing code.
  - **Explain** how to translate repetition in your algorithm into code.
  - **Explain** how to translate decision making in your algorithm into code.
  - **Explain** how to translate mathematical calculations in your algorithm into code.
  - **Explain** how to translate names that you give to values in your algorithm into code.
  - **Explain** how to translate alteration of values in your algorithm into code.
  - **Explain** how to translate your algorithm finding its answer into code.
  - **Explain** how to translate your complex steps (which do not translate directly to a line or two of code) into code.
  - **Explain** what to do if you struggle with Step 5.
  - **Define** *top-down design*.
  - **Explain** how this approach to translating algorithms into code uses the idea of top-down design.
  - **Explain** how this approach to translating algorithms to code helps with abstraction.
  - **Define** *composability*.
  - **Explain** why programming languages are designed such that language features are composable.
  - Note: you should take a minute to go through the two examples at the end of this section and see how the comments translate into code. However, don't get hung up on that, as we will come back to the learning objectives related to *doing* this translation later.

- **A Complete Example**
  - **Recognize** how The Seven Steps were applied in the example video.

- **Next Steps: Compiling, Running, Testing, and Debugging**
  - **State** what steps 6 and 7 of The Seven Steps are.
  - **Define** *source code*.
  - **Explain** (at a high level—more details in Ch 5) what must happen to your source code before you can run it.

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.

Note that many of the learning objectives in this section are going to require a lot of work and practice over the long term. You are not going to master them immediately. Instead, you should start to practice with them on the examples in this chapter. We have marked learning objectives with one star (*) if you are going to need significant practice, but should have fairly good mastery by the end of the assignments associated with this chapter. We have marked learning objectives with two stars (**) if we expect you to make a good start on them with reading and practice here, but that you will need to work on them well beyond this chapter.

- **Step 1: Work an Example Yourself**
  - **Work*** examples of problems yourself.
  - **Analyze**** causes of an answer being too obvious and **evaluate**** ways to remedy that problem
- **Step 2: Write Down What You Just Did**
  - **Assess*** how the data involved in your problem should be represented as numbers.
  - **Break down**** what you did in Step 1 in a step-by-step fashion.
  - **Formulate*** a clear step-by-step approach to the *particular* instance of the problem you did in Step 1.
- **Step 3: Generalize Your Steps**
  - **Analyze*** how a value depends on the inputs and **produce** a formula for that relationship (generalize values).
  - **Justify*** your value generalization (answer "why") to reduce mis-generalizations.
  - **Categorize** steps of an algorithm into "almost repetitive" patterns.
  - **Assess**** when starting with an identity element will help make repetition more clear.
  - **Differentiate**** under what conditions some steps happen and some steps do not (generalizing conditionals).
  - **Formulate**** a condition to make decisions of whether steps happen (generalizing conditionals + generalizing values).
  - **Rewrite**** almost repetitive steps to be repetitive.
  - **Select**** the next strategy/portion of your algorithm to generalize.
- **Step 4: Test Your Steps**
  - **Select**** test cases which are more likely to help you detect mis-generalizations early.
- **Step 5: Translate Your Algorithm To Code**

- **Select*** the appropriate types to represent your data (which might be built-in, or ones you create).
- **Produce** code to declare any custom types you need (note: this LO was covered in the previous chapter, but we re-iterate it here).
- **Assess** when steps call for a looping construct and **select** the appropriate loop type (for, while, do-while) for those steps.
- **Produce*** code which correctly uses a loop to accurately reflect the repetition described in a generalized algorithm.
- **Assess** when steps call for a decision making construct and **select** the appropriate decision construct (if/else or switch/case) for those steps.
- **Produce*** code which correctly uses a decision construct to accurately reflect the decision described in a generalized algorithm.
- **Produce*** code which correctly translates mathematical calculations into C expressions.
- **Produce*** code which correctly declares, initializes, assigns to, and uses variables based on the use of named values in the generalized algorithms.
- **Produce*** code which correctly returns a value when the generalized algorithm indicates that the answer is known.
- **Evaluate*** when the steps in a generalized algorithm are too complex for direct translation to code (and thus you should abstract them into another function).
- **Produce** a call to a to-be-written function when complex steps need to be abstracted out.
- **Recognize** that you need to do The Seven Steps on whatever complex problem you abstracted out.
- Note: top-down design as we are doing it now is captured in the above learning objectives. We will revisit top-down design in more depth in Chapter 13.
- **Use*** composability to apply any of the techniques in this section nested inside of other constructs (loops inside of loops, ifs inside of loops, etc).
- Note: at this point you should make sure you can go through the two examples at the end of this section with complete understanding (*i.e.*, you should be able to do programming tasks of comparable difficulty from Step 1–5.

- **A Complete Example**
  This section does not have any new learning objectives for a second reading, but rather shows examples of the LOs covered in Steps 1–5. We recommend trying this example problem before re-watching the video.

3 Key Learning Objectives

The top-level learning objective for this chapter is **produce** code to solve a problem. As we discussed in the second reading, many of the learning objectives that we covered there will require a lot more practice than just reading about them and seeing a few examples. All of these learning objectives work together to build up your ability to write code for a problem statement.