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.

- **Pointers: Intro**
  - This chapter starts with `naive_swap`. You should already have the skills to **execute** it (or similar code) by hand and **explain** why the output is a = 3, b = 4 (and not a = 4, b = 3).
- **Pointer Basics**
  - **Define** *pointer*
  - **Explain** the conceptual representation of a pointer.
  - **Define** *type constructor*.
  - **Explain** how to create types with pointers in C.
  - **Explain** what happens when you assign to a pointer in C.
  - **Define** *address-of operator*.
  - **Explain** what `&someVariable` does in C, and how to draw a digram of it.
  - Recall some Chapter 2 Learning Objectives: **Define** *lvalue* and **explain** what kinds of things are lvalues (note: we are about to add more things that can be lvalues).
  - **Define** *unary* operator.
  - **Define** *dereference* (a pointer).
  - **Explain** what `*somePointer` does in terms of executing code by hand.
  - **Explain** the two different uses of `*` related to pointers, and how to tell the difference between them.
  - **Execute** the code in Figure 8.1 by hand.
    - Note: we usually reserve working the examples and higher-level learning objectives for the second reading, however, Figure 8.1 shows several fundemental operations on pointers that you will need to understand before moving on to the rest of the chapter.
- **A Picture Is Worth a Thousand Words**
  - **Recognize** that *drawing pictures* is critical to understanding code with pointers.

    - Note: Drawing pictures is REALLY important with pointers. Going forward, drawing careful pictures will only become more and more important.
  - **Explain** how to draw a variables who types is a pointer to something.
  - **Explain** the new kind of lvalue that pointers introduce.
  - **Explain** how to execute assignment statements by hand, even if they are complicated and involve pointers.
  - **Execute** the code in Video 8.2 by hand.
    - Note: we usually reserve working the examples and higher-level learning objectives for the second reading, however, Video 8.2 shows several fundemental operations on pointers that you will need to understand before moving on to the rest of the chapter. We recommend doing Video 8.2 yourself first, then watching to make sure you did everything correctly (And if not, understand what you should have done instead). If you struggle with Figure 8.1 and Video 8.2, we recommend pausing (get a snack, take a walk, etc—come back in 20 minutes) and go back through these first sections again before proceeding.

- **swap, Revisited**
  - **Explain** how pointers allow us to write a correct implementation of `swap`.

- **Pointers Under the Hood**
  - **Define** *address*
  - **Explain** how data types which require more than one byte are stored in memory.
  - **Explain** why the size of a pointer does not depend on the type it points at.
  - **Define** *addressable data*.
  - **Explain** why a pointer can only point to addressable data.
  - **Explain** the relationship between lvalues and addressable data.
  - **Recognize** that each program has the entire address space
    - Note: this is often strange and confusing to novice programmers, as many programs run at once. If you take a class which explains how hardware works, you will learn about *virtual memory*, where the hardware and OS work together to give ever program its own *virtual address space*, while managing *physical memory*. However, how that works is beyond the scope of this class, and is not part of your learning objectives.
  - **Explain** how the code for a program is stored in memory.
  - **Define** *static data*.
  - **Recognize** that the *heap* is a region of memory, and that you will learn about it later.
  - **Explain** where and how the stack frames (that you have been drawing since Chapter 2) are stored in memory.
  - **Define** *calling convention*.
    - Note: specifics of calling conventions are beyond the scope of this class.

- **Recognize** that programs may not access address 0.
- **Define** *null pointer*.
- **Explain** why null pointers can be useful.
- **Explain** how to draw null pointers in diagrams.
- **Explain** what happens if you attempt to dereference a null pointer.
- **Define** *segmentation fault*.
- **State** the type of `NULL` in C.
- **Define** *void pointer*.
- **State** which operations you cannot do on a void pointer (that you can do on other pointers).
- **Explain** why you cannot do those operations on a void pointer.
- **Explain** where/why you need to check for NULL in your code.

- **Pointers To Sophisticated Types**
  - **State** the order of operations for `*` and `.`
  - **Explain** when `*v.f` is appropriate, and how it would evaluate.
  - **State** where parentheses are needed in `*p.f` if `p` is a pointer to a struct, and you wish to access field `f` in it.
  - **State** what the `->` operator does, and how to desugar it into `*`, `.` and `()`.
  - **Explain** a type with multiple *s in it (*e.g.*, what is an `int **`).
  - **Recognize** that the rules for pointer operations are unchanged for pointers to pointers.
  - Note: not truly a learning objective, but an important bit of context is to make sure you know we will see a lot of uses of pointers through the rest of this course. You need to be proficient in their mechanics before we can use them for complex tasks.
  - **Explain** how you would figure out the type of the expression `&someLValue`
  - **Explain** how you would figure out the type of the expression `*somePointer`
  - **Explain** how `*` and `&` are inverse operations (and what that means).
  - **Define** *const*.
  - **Explain** why it can be useful to declare data as const.
  - **Explain** how to interpret const qualifiers on pointer to pointers.
  - **Explain** what happens if you make a const pointer which points at a variable that was not declared const (in terms of what can and can't be changed)
  - **Explain** the compiler error "intiialization dsicards 'const' qualifier from pointer target type"

- **Aliasing: Multiple Names for a Box**
  - **Define** *alias*.
  - **Explain** how aliasing relates to Step 2 of The Seven Steps.

  - **Explain** how aliasing can cause values to change in non-obvious ways.
  - **State** what gdb command can help you debug situations in which values change unexpectedly due to aliasing.
  - **Recognize** that it is generally a bad idea to have a pointer of the wrong type pointing at a piece of data.
    * Note that there are circumstances where such things are appropriate, but they will not come up in this course, and are generally quite advanced. Also, knowing exactly what is going on/understanding why requires a deeper knowledge than we are going into.

- **Pointer Arithemetic**
  - **Define** *pointer arithemetic*.
  - **Explain** why, when compiling `somePointer + n`, the compiler multiples `n` by the number of bytes required to hold the type of `*somePointer` before adding.
  - **Define** *undefined behavior*.
  - **Explain** why you need a sequence of consecutive boxes for pointer arithmetic to be useful.

- **Use Memory Checker Tools**
  - Recognize that `valgrind` can be a useful tool to help debug pointer-related problems.
  - Recognize that `-fsanitize=address` can be a useful tool to help debug pointer-related problems.

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.

- **8.1–8.3**
  - **Execute** code with pointers (including declaration, initialization, address-of, changing the pointers, and dereferencing pointers)
  - **Determine** if an expression is an lvalue

- **Pointers Under the Hood**
  - **Show** how data (including pointers) are represented numerically in memory
  - **Execute** code by hand using the numerical representation of data stored in memory (and the corresponding addresses).

- **Pointers to Sophisticated Types**

  - **Evaluate** expressions with `*`, `.` and `->` using the correct order of operations.
  - **Use** `->` correctly in code you write (and don't use `(*a).b`).
  - **Execute** code that uses structs and pointers mixed together (structs with pointers, pointers to structs, etc).
  - **Execute** code that uses pointers to pointers[1]
  - **Compute** the type of an expression involving pointers to pointers.
  - **Compute** whether an expression names a "const" box, given declarations which may include pointers to pointers.
  - **Connect** the ideas that "pointer is a type constructor" with the idea that we can make pointers to pointers.

- **Aliasing: Multiple Names For Boxes**
  - **Calculate** all of the names for a box (in a diagram of program state) that has aliases.
  - Note: over time you will need to work on **Select** which name for an aliased box to use in your algorithm. However, you will not have practice with that by the end of the second reading of this chapter.

- **Pointer Arithemetic**
  - You will do a lot more with this in the next chapters. Right now, we just want to introduce the high-level idea.

3 Key Learning Objectives

This chapter focuses on the *mechanics* of pointers. Note that we do not actually have an **create**-level learning objectives (and the only **evaluate**-level we mentioned was referencing something to work on over time). This lack of top-level learning objectives is unusual, but pointers are a big, complicated subject. Accordingly, we need to introduce them to you with a focus on their concepts and mechanics. After you become proficient in this material, we will move on to arrays (which use pointers) in the next chapter, and have **create**-level learning objectives to write code with arrays (and thus pointers).

For now, the most key learning objective of this chapter is:

- **Execute** code by hand with pointers (including pointers to pointers, pointers to structs, structs with pointers, `->`, `&`, and `*`).

[1] Whenever we say "pointers to pointers" in these learning objectives, we mean of any number of layers of pointers, *e.g.*, pointers to pointers to pointers to pointers are included.