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.

- **Motivating Example**
  - Note: the following five things are not really core learning objectives of this class, but are good for your general computational knowledge:
    - **Define** *Caesar cipher*
    - **Define** *Vigenere cipher*
    - **Recognize** that classical ciphers are easily broken and cryptography should only be done with algorithms devised by experts in the field.
    - **Define** *frequency counting*
    - **Explain** how frequency counting can break Caesar ciphers.
  - **Explain** why it quickly becomes impractical to start each piece of data in its own variable, as the amount of data increases.
- **Array Declaration**
  - **Define** *array.*
  - **State** the two pieces of information required to create an array.
  - **State** the syntax for declaring an array.
  - **Draw** a diagram of an array variable, given its declaration.
  - **Explain** why an array variable does not have a box of its own.
  - Note: there is so nastiness around the fact that the C-standard says array names are *lvalues* despite the fact that they don't act like any other lvalue. The only reason they are called lvalues is because you can do `&myArray` but even that doesn't work like other array names (and is used incredibly rarely). You do NOT need to get hung up on the technical details here. myArray is technically an lvalue (C standard), but really shouldn't be (doesn't act like one).
  - **State** the syntax for initializing an array as part of its declaration.
  - **Explain** what happens if you provide the wrong number of elements in the initializer of an array.
  - **Explain** what happens if you omit the array size, but provide an initializer.
  - **State** the syntax for initializing a struct in the same line as it is declared.

- **State** the variation of the struct initialization syntax to be explicit about field names.
- **Explain** why being explicit about field names in a struct initializer is preferable.
- **Recognize** that array variables have an array type including their length
- **Recognize** that in almost all situations, the type of the array decays to a pointer
- **Define** *decay* (of array types).
- **Recognize** that there are a few special circumstances where the array type does not decay.
  - Note: two of these will come later. For the third one that is explained here, you really don't need to get hung up on it. We give it for completeness, but it is not used explicitly terribly often.

- **Accessing an Array**
  - **Explain** how you can access the elements of an array with pointer arithmetic
  - **Define** *indexing* (an array).
  - **State** the syntax for indexing an array in C.
  - **Explain** why `someArray[someIndex]` can be used as either an lvalue or an rvalue.
  - **Define** *zero-based indexing.*
  - **Explain** what happens in C when you attempt to index an array beyond its bounds.
  - **Recognize** that array indexing is equivalent to pointer arithmetic and dereferencing
    - Note: while they are equivalent, indexing is often easier to read, and thus preferable.
  - **Explain** the equivalence between `&myArray[i]` and `myArray + i`

- **Passing Arrays as Parameters**
  - **State** the parameter type to declare for a function which needs to receive an array as a parameter.
  - **Recognize** that there is no way to properly derive the length of an array just from the array itself in C.
    - **Note:** some of you may find something on StackOverflow which tells you that you can use `sizeof`. It might even give you code that "shows" you it works. Note that `sizeof` is a compile-time operator, and looks at the type of its operand. It is one of the few special cases that will not cause array-to-pointer decay and will ONLY give the actual size of the array if the type has not already decayed. The `sizeof` operator was not intended for computing array lengths, and should not be used in this way. Even if you use it in a case where the type has not decayed, it makes your code fragile: abstracting that region out into a function will break it.
  - **Recognize** that you need to explicitly pass the length of an array to functions which need that information.
  - **Explain** why array indexing works properly on parameters with pointer type that actually point at arrays.

    - **State** the alternate syntax for passing an array to a function, and why some people prefer it.
    - **Recognize** that even if you put a size in the square brackets of a function parameter, the compiler does not check it.

- **Writing Code with Array**
    - **Recognize** that the most common pattern in accessing an array is to access the elements in order.
    - **Recognize** that other patterns may exist, and you have to find them.
    - **Recognize** that drawing a picture is incredibly important, especially if you have complicated data.
    - **Define** *dangling pointer*.
    - **Explain** why returning the address of a local variable (including returning an array) results in a dangling pointer.
    - **Explain** why sometimes use of a dangling pointer may not result in any discernible problems, but is still problematic.

- **Sizes of Arrays**
    - **Define** `size_t`.
    - **State** the most correct type to use for sizes or indices of arrays.
    - **Explain** why `size_t` is more appropriate than `int` for the size and indices of an array, under what circumstances the distinction does not matter, and under what circumstances the distinction does matter.
    - **State** what the `sizeof` operator does, and what type it evaluates to.
    - **Recognize** that `sizeof` is evaluated at compile time, based only on types.
    - **Recognize** that `sizeof` is one of the special cases where array-to-pointer decay does not happen.
    - **Recognize** that you should NOT use `sizeof` to try to get the length of an array.

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.

- **Array Declaration**
    - **Execute** code by hand with array declarations, properly drawing diagrams of the arrays.
    - **Execute** code by hand with array and/or structs initialized in their declarations.
    - **Calculate** the type of an expression involving arrays.

  - **Execute** code by hand where an array variable is assigned to a pointer.

- **Array Declaration**
  - **Execute** code by hand with pointer arithmetic.
  - **Execute** code by hand array indexing.
  - **Convert** between array indexing and pointer arithmetic plus dereferencing.

- **Passing Arrays As Parameters**
  - **Execute** code by hand where arrays are passed to functions.

- **Writing Code with Array**
  - **Determine** the names of boxes involving arrays.
  - **Select** from multiple names for a box, the one which is most appropriate for generalizing your algorithm.
    - Note: this LO will need practice beyond the second reading, and mastery beyond this chapter.
  - **Determine** when values used in Step 2 (of The Seven Steps) are the result of indexing into an array.
  - **Determine** patterns in array indexing (value generalization).
  - **Write** code using arrays.
  - **Execute** code by hand where a pointer becomes dangling, and **determine** the cause of that dangling pointer.

- **Sizes of Arrays**
  - **Assess** when `size_t` is the most appropriate type for a variable or parameter.

3 Key Learning Objectives

Overall, you want to become proficient programming with arrays. Your core learning goals—both from reading, and practicing in upcoming assignments—are:

- **Execute** code by hand with arrays and pointers.
- **Assess** if arrays are the appropriate way to store data for your problem.
- **Write** algorithms and code with arrays.
- **Use** valgrind to find array and pointer related problems in your code.
- **Use** gdb to debug code with pointers and array.

We note that the last two were not explicitly discussed, but are applications of tools you have learned previously. Code with arrays and pointers are where these tools are really going to start showing their benefit, and you will want to practice with them and become proficient.