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.

- **Dynamic Allocation:Intro**
  - **Recall** what a *dangling pointer* is (from Chapter 9).
  - **Explain** why returning an array that is a local variable is problematic (review from Chapter 9).
  - **Define** *dynamic memory allocation*.
  - **Define** *heap*.
  - **Explain** the major differences between allocating memory on the heap and the stack.
  - **State** the C library function to allocate memory on the heap.
  - **State** the C library function to free memory on the heap.
  - **State** the C library function to resize a block of memory on the heap.
  - **State** the C library function to read an arbitrary length string from a file, dynamically allocating memory for it.
- **malloc**
  - **Explain** how to call `malloc` and what it does.
  - **Recall** what a `void *` is.
  - **Explain** why `malloc` returns a `void *`.
  - **Recall** what *portability* means (from Chapter 5).
  - **Define** *maintainability*.
  - **Explain** two reasons why you should not assume how large a particular type is when calling `malloc`.
  - **Recall** what the `sizeof` operator does (from Chapter 9).
  - **Explain** how to use `sizeof` to correctly calculate the parameter to pass to `malloc`.
  - **Explain** why the *best* way to use `sizeof` when calling `malloc` is to put an expression which dereferences what the result will be stored in, rather than putting the type as the operand of `sizeof`—*e.g.*,

    ```
    thing * lhs = malloc(10 * sizeof(*lhs));
    ```

- **Explain** how to draw a diagram of program state with memory allocated in the heap (*e.g.*, by `malloc`).
- **State** what `malloc` returns if it fails.
- **Explain** how `malloc` can be used to write functions which build and return data structures.
- **Recall** what *composability* is (from Chapter 4).
- **Explain** how *composability* relates to the idea of building complex data structures with `malloc`.
- **Recall** that assignment (`=`) on pointers just changes where the pointer on the left side of the `=` points, it does not copy anything (from Chapters 8 and 10).
- **Define** *shallow copy*.
- **Explain** the aliasing that exists in the shallow copy example in Figure 12.4, and what impact it has on use of those data structures.
  - Note: on the second reading, we will expand this LO to be more general, but on the first reading you should be able to explain it for this particular example.
- **Define** *deep copy*.
- **Explain** why there is no aliasing between an original structure and a deep copy of it.

- **free**
  - **Explain** how memory on the stack is freed.
  - **Explain** how memory on the heap is freed.
  - **Explain** the parameter to `free`, including its type and what kind of values are appropriate to pass to it.
  - **Explain** what happens to a region of memory after you `free` it, and how you should reflect that effect when executing code by hand.
  - **Explain** why `q` is also dangling in Video 12.2
    - Note: on the second reading, we will expand this LO to be more general, but on the first reading, you should be able to explain it for this particular example.
  - **Explain** how you need to approach freeing complex structures.
  - **Define** *memory leak*.
  - **Explain** under what circumstances memory leaks may not be important and why not.
  - **Explain** under what circumstances memory leaks may be quite problematic and why.
  - **State** that you are going to be sure to free all the memory you allocate.
  - **Recognize** the `valgrind` output that indicates you have freed all memory and certainly do not have any memory leaks (which is the goal for all your assignments in this class).
  - **Explain** how the example in Video 12.3 leaks memory, how you can recognize that the memory was leaked, and how the code was fixed.
    - Note: on the second reading, we will expand this LO to be more general, but on the first reading, you should be able to explain it for this particular example.

  - Note: 12.2.2 doesn't introduce new learning objectives, it just is aimed at helping you understand the above.
  - **Define** *double freeing*.
  - **Explain** what happens when you double free memory in your program, and how to make debugging those problems easier.
  - **Explain** *why* you must `free` exactly the pointer returned by `malloc` (and not some other pointer into the middle of the block).
  - **Recognize** that `free`ing memory that is not on the heap causes problems.

- **realloc**
  - **Explain** what `realloc` does
  - **Explain** how to properly call `realloc`, including what the parameters mean, and how to use the return value.
  - **Explain** why `realloc` may or may not give you back the same pointer you passed to it.
  - **Explain** what happens if `realloc` fails.
  - **Explain** how `realloc` is being used in Video 12.5
    - Note: on the second reading, we will expand this LO to be more general, but on the first reading, you should be able to explain it for this particular example.

- **getline**
  - **Explain** why you need dynamic allocation to read a line of input of unknown length from a file.
  - **Explain** why the first and second parameters to `getline` have types `char **` (instead of `char *`) and `size_t *` (instead of `size_t`).
  - **Explain** what happens when you call `getline` when `*linep` is NULL.
  - **Explain** why your program will segfault if you call `getline` when `linep` is NULL.
  - **Explain** what must be true of `*linecapp` when you call `getline` and `*linep` is not NULL.
  - **Explain** what happens when you call `getline` when `*linep` is not NULL.
  - **Explain** how Video 12.6 uses `getline` to read every line in a file, and correctly `free` the memory.
    - Note: on the second reading, we will expand this LO to be more general, but on the first reading, you should be able to explain it for this particular example.
  - **Explain** how Video 12.7 uses `getline` and `realloc` together to read every line in a file and store them in an appropriately sized array.
    - Note: on the second reading, we will expand this LO to be more general, but on the first reading, you should be able to explain it for this particular example.

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.

- **malloc**
  - **Execute** code by hand with calls to `malloc`, correctly drawing the allocation in the heap (outside of any frames).
    - Note, this includes more complex structures, such as the polygon/point example
  - **Assess** if dynamic allocation is appropriate for code you need to write (considering, among other things, aliasing and freeing memory).
  - **Use** `sizeof` correctly (best is `*lhs` where lhs is what you are assigning to) to specify how many bytes to allocate with `malloc` (or `realloc`)
  - **Select** a shallow or deep copy appropriately for the code you are writing.
  - **Illustrate** the aliasing in shallow-copied data structures, and **determine** its impacts in the code.
  - **Categorize** a drawing of program state as showing an object having been shallow copied, deep copied, or neither.
  - **Write** code to shallow copy a given data structure.
  - **Write** code to deep copy a given data structure.
- **free**
  - **Execute** code by hand with calls to `free`
  - **Show** which pointers become dangling pointers as the result of a call to `free`.
  - **Categorize** incorrect `free`-related problems (memory leaks, double free, freeing memory not returned by malloc, freeing memory not on the heap, use after free) when executing code by hand.
  - **Determine** where a call to `free` should be inserted (and what pointer should be freed) to fix a memory leak.
  - **Determine** what call(s) to `free` should be removed or changed to fix a double free
  - **Determine** how to change a call to `free` when it is freeing memory not exactly returned by `malloc`.
  - **Assess** whether to move or remove calls to `free` to fix use after free errors.
  - **Determine** where to move calls to `free` (when appropriate) to fix use after free errors without introducing other problems.
  - **Use** `valgrind` to identify memory leaks in your code.
  - **Write** code which correctly uses `free` to free all dynamically allocated memory.
- **realloc**

  - **Execute** code by hand with calls to `realloc`
  - **Determine** when code improperly assumes `realloc` leaves the memory in place, when executing code by hand.
  - **Assess** when `realloc` is appropriate to the programming task/algorithm you have.
  - **Write** code which correctly uses `realloc` to resize blocks of dynamically allocated memory.

- **getline**
  - **Execute** code by hand with calls to `getline`
  - **Select** from different methods of reading input, including `getline` (and the others from Chapter 11).
  - **Write** code which correctly uses `getline` to read input simply (as in Video 12.6)
  - **Write** code which correctly uses `getline` to read input into more complex structures (as in Video 12.7)

3 Key Learning Objectives

These are the biggest learning objectives of this chapter, many of which will require practice beyond this reading:

- **Execute** code which uses dynamic memory allocation (including `malloc`, `free`, `realloc`, and `getline`)
- **Categorize** free-related problems when executing code by hand.
- **Use** valgrind to identify memory-related problems in code.
- **Use** gdb to help debug memory-related problems in code.
- **Write** code which uses dynamic memory allocation (including `malloc`, `free`, `realloc`, and `getline`)