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.

- **The Operating System**
  - **Define** *operating system.*
  - **Define** *system call.*
  - **Explain** the relationship between a program, the C library, the OS, and the hardware.
  - **Explain** why system calls might fail.
  - **Explain** what happens when a system call fails, and how your program could find out why the call failed.
  - **Name** the C library function which prints a descriptive message about the current value of errno.
  - Note: AoP alludes to the fact that there is a LOT more to learn about OSes. If you are an ECE student taking ECE 650 in the future, you will learn a significant amount about OSes there.
- **Command Line Arguments**
  - **Define** *command line arguments.*
  - **State** how to declare `main` so that you can access the program's command line arguments.
  - **Explain** what `argc` and `argv` mean and how you use them.
  - **Explain** what kinds of error checking you should consider when processing command line arguments.
  - Note: it is great to know that `getopt` is available for complex option processing, but we aren't going to focus on it.
  - **Define** *environment.*
  - **Recognize** that programs can access the `variable=value` pairings in their environment.
    - Note: we aren't going to make use of this, but it might come up later in your programming career
  - **Name** the system call which makes an identical copy of the current process.
  - **Name** the system call which replaces the current program with a requested program.

- **Explain** what happens when a new program is created with fork and exec.
- **Explain** what happens when main returns.

- **Files**
  - **Recall** what a *file* is (from Appendix B).
  - **Define** *stream*.
  - **Name** the type in C which is associated with a stream.
  - **Name** the C library function and which opens a file.
  - **Recall** what a *pathname* is (from Appendix B).
  - **Explain** why we do not need to know the exact details of a FILE struct.
  - **Define** *file descriptor*.
  - **Explain** why `fopen` might fail, and what happens when it does.
  - Note: you will use “r” and “w” all the time for file modes, and look up the others as needed.
  - **Name** the C library function to read one character at a time from a file.
  - **Define** *EOF*.
    - Note: many students mistakenly think that EOF is a character that is in the file. EOF is a value that is returned to indicate that no more characters exist in the file.
  - **Explain** why `fgetc` returns an int, not a char.
  - **Explain** what challenges are posed in writing loops to read a file by the fact that each call to `fgetc` advances the position in the stream.
  - **Explain** what “an assignment is also an expression that evaluates to the value that is assigned” means.
  - **Explain** the commmon idiom for how to structure loops which read input.
  - Note: the part about the subtle bug if you use a char to hold the return value of `fgetc` is a bit more advanced. You should read it, but not get hung up on it now—instead, we’ll revisit that on the second reading.
  - **State** that there are many useful C library functions for determining what kind of character something is.
  - **Name** the C library function that reads one line of input from a file, up to a maximum length.
  - **Explain** how to use `fgets`.
  - **Recall** that you should NEVER use `gets` (from Chapter 10).
  - **Name** the C library function to read a fixed amount of data from a file.
  - **Explain** how to use `fread`.
  - **Explain** why we recommend against using `fscanf`, insted preferring another method to read input, and use of `sscanf`.
  - **Name** the C library function to print formatted output to a file.

- **Name** the C library function to print a single character to a file.
- **Name** the C library function to print a literal string to a file with no format conversion.
- **Name** the C library function to print non-textual data to a file.
- **Explain** why some write failures may not be detected immediately.
- **Name** the C library function to close a file.
- **Explain** what to do when fclose fails (and why it is a bit of a difficult question).

- **Other Interactions**
  - **Explain** why part of the UNIX philosophy is "make everything look like a file."
  - **Define** `stdout`.
  - **Define** `stdin`.
  - **Define** `stderr`.
  - **Explain** how C and UNIX make interacting with the terminal "look like a file."
  - **Define** *socket*.
  - **Define** *pipe*.
  - **Define** *device special file*.
  - **Explain** some interactions with the OS that do not look like a file, and how those work.
  - **Define** *asynchronously*.
  - **Define** *signal*.
  - **Explain** what happens when your program segfaults.

## 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.

- **The Operating System**
  - **Write** code which properly checks for failures of system calls (or library functions which use system calls), and describes the error appropriately with `perror`.
  - As noted above, the details of the OS are out of scope of this class. Depending on your program/track, you might take ECE 650, and possibly more electives around OSes to go much deeper. However, every programmer should have some basic conceptual understanding and terminology. We still recommend re-reading this section, but most higher level learning objectives are left to other courses.

- **Command Line Arguments**
  - **Draw** a diagram of the initial state of the program (on entry to `main`) when the program is passed command line parameters, and `main` takes `argc` and `argv`.

  - **Determine** what errors your program should check for in processing its command line arguments.
    - Note that this also implies that you should **construct** appropriate test cases for each error you handle based on what you learned in Chapter 6.
  - **Write** programs which take command line arguments and use them appropriately.
- **Files**
  - **Execute** code by hand which involves FILEs, including `fopen`, `fgetc`, `fgets`, `fprintf`, and `fclose`.
    - Note: `fread`/`fwrite` are good to know about, but we don't practice with them in this course.
  - **Write** code which involves FILEs, including `fopen`, `fgetc`, `fgets`, `fprintf`, and `fclose`.
  - Note: not a learning objective for this course, but for your longer term trajectory, you should learn to **assess** how to handle failures from system and library calls, and **implement** those solutions in your code.
- **Other Interactions**
  - **Execute** code which reads stdin or writes stdout/stderr.
  - **Write** code which reads stdin or writes stdout/stderr.
  - **Assess** whether a particular piece of output should be written to stdout or stderr.

## 3 Key Learning Objectives

The most key learning objectives from this chapter (Which will require practice after you finish reading) are:

- **Draw** a diagram of the initial state of the program (on entry to `main`) when the program is passed command line parameters, and `main` takes `argc` and `argv`.
- **Write** programs which take command line arguments and use them appropriately.
- **Execute** code by hand which involves FILEs, including `fopen`, `fgetc`, `fgets`, `fprintf`, and `fclose`.
- **Write** code which involves FILEs, including `fopen`, `fgetc`, `fgets`, `fprintf`, and `fclose`.