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.

- **Strings**
  - **Explain** the differences between strings in C and strings in C++.
  - **State** how to find the length of a string in C++.
  - **State** how to index a string in C++, including what type the operator returns.
  - **State** how to create a C++ string from a C string (including a string literal).
  - **State** where to find information about more methods, constructors, and operators that C++ strings have.
  - **Explain** why code with strings (or other objects) might look tail recursive but actually not be.
  - **Explain** why copying objects may cause large performance problems.
    - Note: while we don't focus on performance, we do put timeouts on the graders—if something takes much much much longer than it should, the grader assumes its in an infinite loop and kills the program. Excessive needless copies make a program very slow in general, but the overhead of the copies is even worse when running in valgrind. The most common cause of submissions that do not have infinite loops timing out in the grader is excessive copying.
- **Output**
  - If needed, you might review anything from Ch 11 to make sure you can **explain** how IO works in C, though you should be pretty familiar with that by now.
  - **Explain** why C++ IO is designed very differently from C's IO.
  - **State** the most typical class and overloaded operator used to write output in C++, as well as the relevant header file.
  - **State** the C++ equivalents of stdout and stderr.
  - **Explain** how to use the stream insertion operator.
  - **State** the return type of the stream insertion operator and **explain** why this type is appropriate. (Note: a corrolary of this is **explain** how multiple insertion operators chained together, such as `a << b << c` works).
  - **State** the parameter types for an overloaded stream insertion operator.

- **Explain** why the stream insertion operator must be declared outside of the class rather than as a member.
- **Explain** the `friend` keyword in C++.
- **Explain** why you want to use `friend` rarely.
- **Explain** why it is ok to use `friend` for overloaded stream insertion operators.
- **Explain** how C++'s approach to output aligns with the OO-principles you have learned so far.
- Note: *manipulators* can be really useful, but are not a key focus of this course. Its good to read about them, know they are there, and look them up/come back to this section if you need them for something.

- **Input**
  - **State** the class and overloaded operator used for *formatted* input in C++.
  - **State** the C++ equivalent of stdin.
  - **Explain** why, when writing the stream insertion operator, the second parameter is typically a const reference, while for the stream extration operator it is typcally a non-const reference.
  - **Explain** how to check for and clear errors on C++'s input streams.
  - **State** the behavior of the stream extraction operator on strings.
  - **State** the C++ function to read an entire line of input (up to a newline).
  - Note: Drew is not a big fan of using `>>` to read input. If you have any error checking more robust than just "did it all work or anything go wrong" the checking code gets quite cumbersom, and fine-grained recovery (figuring what failed, where/why) can be complex. Drew prefers to read a while line with `std::getline`, then take it aparet. If stream extraction is desirable, doing it on a string stream (coming up in Section 16.4.2) may be preferable as your error recovery code can then look at the entire line that was read.

- **Other Streams**
  - **State** the classes to use in C++ to read from and write to files, as well as the relevant header file to use.
  - **State** the method on C++'s input and output stream classes to close a file when you are done with it.
  - Note: for now you should recognize that there are some compatible types that can be used where another type is expected (*e.g.*, you can used an `ofstream&` where an `ostream&` is expected). However, truly understanding this requires waiting until Chapter 18.
  - **Explain** what a `stringstream` is and how to use it.

2 Second Reading

Note that this chapter is very different from all of the others, in that we are going back to old ideas (strings and IO) and just showing you how they are done differently in C++. Accordingly, we have a lot less to show you about the basic mechanics of things—we assume you already know how strings work and have a lot of practice reading and writing code with them.

This puts you (and us) in an unusual situations for this course—but a pretty common situation in the real world. You have the higher level skills (reading code with strings and IO, writing code with strings and IO), but are not putting in *different* lower-level knowledge to build those skills on (swapping out the particular methods, mechanisms, etc). Such a situaion comes up moderately often in the real world: as you learn a new programming language, start using a different library or framework to do something, or change to a different tool. You'll often have the programming skills to do what you need and just need to "shift" those higher-level learning objectives onto different lower-level knowledge. We'll note this is a skill you should work on over the long term.

You should still sleep on this chapter, and come back and do a second reading—during that second reading, you should think carefully about how you can "relearn" your higher-level strings and IO skills for C++.

- **Find** when copying is being done during execution of C++ code by hand, and **determine** whether that copying is needed or excessive.
- **Categorize** C++ code as head or tail recusive with an awareness of object destruction (Note: this isn't really new, but rather is just bringing together learning objectives from Ch 7 and Ch 15).
- **Convert** `printf` to stream extraction.
- **Execute** C++ code by hand that includes stream extraction, stream insertion, std::cin, std::cout, std::cerr, ifstream, ofstream, stringstream and C++ strings.
- **Write** C++ code using stream extraction, stream insertion, std::cin, std::cout, std::cerr, ifstream, ofstream, stringstream and C++ strings.

3 Key Learning Objectives

As is typical, **executing** code by hand and **writing** code are the most key skills in this chapter. However, as we mentioned earlier, there is a bit of a "meta" learning objective (learning how to learn) here of "relearning" how to do familiar programming tasks in a different language.