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.

- **Transition to C++: Intro**
  - **Explain** (at a high-level) the differences between C and C++.
  - **Explain** (at a high-level) why we are not working on C++11, C++14, C++17, or C++20.
  - **Explain** why C++ has terminology that might be different from most other OO languages, and why we aim to present both sets of terminology.
  - Note: we want to underscore that this class focuses on *programming* and we are learning more about OOP and less about the nasty language-specific details of C++. You'll have enough C++ to write substantial programs in C++, but if, after this course you want to dive deeper into C++, the "Effective C++" series by Scott Meyers (`https://www.aristeia.com/books.html`) could be a good next step.
- **Object-Oriented Programming**
  - **Define** *object-oriented programming*.
  - **Explain** the similarities and differences between classes and structs.
  - **Explain** what "objects are instances of classes" means.
  - **Define** *member function* (C++ term) / *method* (used in most other OO languages).
  - **State** the rule change in C++ for declaring and using structs (compared to C).
  - **State** the syntax for declaring a class.
- **Access Control**
  - **Define** *access control*.
  - **Explain** public access.
  - **Explain** private access.
  - **State** the default access in a struct.
  - **State** the default access in a class.
  - **Explain** why you would want to restrict access to members of a class.

    - Note: we talked about *invariants* briefly in Chapter 6. We don't delve deeply into them (as they are the province of an Algorithms class). However, we might briefly note that restricting access allows us to maintain invariants more easily. We are able to enforce that `balance>=0` in our BankAccount example. It is not a learning objective here, but for those who are curious, the invariants of a class are what are true of an object of that class after the object is initialized, and after every method call on the object returns.
  - **State** the syntax for access control in C++.
  - **Identify** which members of a class are public and which are private.

- **Encapsulation**
  - **Define** *encapsulation*.
  - **State** the syntax to call methods on objects in C++.
  - **State** what the `this` keyword means in C++.
  - **Explain** the type of `this`.
  - **Recognize** that `this` is the implicit first argument to methods (member functions) in C++.
  - **Explain** what to pass for the `this` parameter when making a method call.
  - **Explain** how `this->` may be used implicitly inside a method.

- **const Methods**
  - **Explain** what `const` at the end of a method declaration (*e.g.*, `int getX() const`) means in C++.
  - **Define** *const-correct*.
  - **Explain** why it is important to write code that is const-correct.
  - **Define** *plain old data* (POD).
    - A lot of students get really hung up on POD vs not-POD. The rules of POD vs not-POD types are very C++ specific, and not the most critical thing to learn.
  - **Explain** why C++ makes the distinction between POD and non-POD types.

- **Static Members**
  - **Define** *static field*.
  - **Explain** why you might use static fields.
  - **Explain** why static methods cannot access non-static members of a class, and why they do not have a `this` pointer.
  - **State** when static fields are initialized.

- **Classes Can Contain Other Types**
  - **State** the two ways that a type can be declared inside of a class.
  - **Define** *inner class*.

- **Define** *scope resolution operator*.
- **State** the syntax of using the scope resolution operator in C++.
- Note: the discussion of escaping private inner types is for completeness and to answer a commonly asked question. Don't get hung up on it: just don't let your private inner types escape.
- **Explain** how code inside of a class (*e.g.*, BankAccount) can make use of a private inner class (*e.g.*, Transaction).
- **Explain** why code inside of an inner class cannot directly access fields or methods of the outer class, and how to deal with this situation.

- **The Basics of Good OO Design**
  - **Explain** what "Classes are nouns. Methods are verbs" means.
  - **Explain** what "Keep classes small and to their purpose" means (Note: in a Software Engineering class, you will expand on this idea with the Single Responsibility Principle).
  - **Explain** what "Be as general as you can be, but no more" means.
  - **Explain** what a "Manager" class is and why you should avoid them.
  - Note: in a Software Engineering class, you will learn a LOT more about good OO design, but that is not only out of the scope of this course, but also requires several ideas you will see later in this course to understand those ideas.

- **References**
  - **State** what a reference is *conceptually*.
  - **List** and **explain** 6 differences between pointers and references.
  - **Explain** how to translate a reference *declaration* to equivalent code with a pointer.
  - **Explain** how to translate a reference *initialization* to equivalent code with a pointer.
  - **Explain** how to translate *a use of* a reference to equivalent code with a pointer.
  - **Recall** that `&` and `*` are inverse operators (from Chapter 8).
  - Note: many students like to get hung up on the question "is a reference an lvalue?" The answer is yes (because it is another name for a box, so it names that box). However, the underlying pointer that the reference is translated to (this translation is not just to help you understand references, but is how code with references is actually compiled) is not an lvalue. That pointer may not have a box of its own.
  - **Recognize** that a const reference may be initialized from something that is not an lvalue.
  - **Explain** how the fact that a const reference may be initialized from a non-lvalue is in conflict with the translation of references to pointers described above.
  - **Explain** how assignment of a non-lvalue to a const reference is implemented (resolving the conflict in the previous LO).
  - **Recognize** that even though references can be translated to pointers (and typically are by the compiler), they are different types, and that difference is significant (with the importance coming soon).

- **Namespaces**
  - Note: we don't really focus on C's static (very different from C++'s static for members of a class). We just mention it for completeness.
  - **Define** *name collision.*
  - **Define** *namespace.*
  - Note: while its good to be able to *e.g.*, **state** the syntax for declaring a namespace, we are not doing anything big enough in this class to need your own namespace, so its not really a learning objective.
  - **State** the namespace in which the C++ standard library resides.
  - **Define** *fully qualified name.*
  - **State** how to use the scope resolution operator to specify a fully qualified name.
  - **State** how to use the `using` keyword to open a namespace.
  - **State** how to use the `using` keyword to bring one particular identifier from a namespace.
  - **Recognize** that you should be wary of opening namespaces, especially in large scopes.
    - Note: we haven't gone into why you should be wary of namespaces here, as we have not introduced overloading yet, but we will discuss it in the next section.
- **Function Overloading**
  - **Define** *function overloading.*
  - **Explain** the rules for legal overloading of a function.
  - **Explain** how the compiler picks the proper overloading of a function for a given function call.
  - **List** and **explain** three reasons why you should be wary of overloading function names.
    - Note: the third of these reasons—the risk of changing the behavior of existing code by providing a different "best overload"—is the worst. It can introduce bugs into parts of the code you are not touching, making them very hard to debug.
  - **Explain** why opening namespaces is a bad idea with regards to function overloading.
  - **Define** *name mangling.*
  - **State** that there is a special syntax required to make use of compiled C code in C++ code.
    - Note: its great if you remember that this is `extern "C"`, but we aren't going to use it. We want you to know that you have to do something in case this ever comes up in your professional careers, but if you just remember "Oh there was that thing in AoP, let me look it up..." that is the goal here.
- **Operator Overloading**
  - **Define** *operator overloading.*
  - **State** the syntax for overloading an operator within a class.

- **Explain** why the `+` operator inside a class looks like it only takes one parameter, and what the second parameter is.
- **Explain** under what circumstances overloading an operator is appropriate.
- Note: When AoP describes `Matrix operator+(const Matrix & rhs ) {`, it should be `Matrix operator+(const Matrix & rhs ) const {`. This will be fixed in some future version.
- **Explain** why `*this` has the appropriate type to return from an operator whose return type is a reference to the class the operator is in (*e.g.*, `Matrix &` in class `Matrix`).
- **Explain** why `operator+=` returns a reference to the class it is in (*e.g.*, `Matrix&`).
- **Explain** why you might want to overload operators which differ only in the `const`-ness of `this`.

- **Other Aspects of Switching to C++**
  - **State** that `g++` is the C++ compiler (instead of `gcc`).
  - **Explain** the similarities and differences in use of booleans in C and C++.
  - **Recognize** that `void *` is not used much in C++ (with better ways to do things coming in future chapters).
  - **State** the differences in header files between C and C++.
  - **Explain** how code should be organized between header files and implementation files in C++.
  - **Define** *default values*.
  - **Explain** how to use default values in C++.
  - **Explain** when it is appropriate to use default values.
  - **State** that `https://cplusplus.com/` has good reference material for C++ (the `https://cplusplus.com/reference/` section is *authoritative documentation*).

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.

- **Object Oriented Programming**
  - **Declare** a class in C++, including appropriate access control declarations for members.
  - **Figure out** the visibility (public or private) of a particular member of a class, given its declaration.
  - **Write** code which calls methods on objects (with `.` and `->`).
  - **Execute** code by hand which calls methods on objects, correctly passing the `this` pointer in your diagram.

- **Figure out** when a name (field or method) implicitly has `this->` on the front of it.
- **Execute** code inside of a class by hand, which makes use of the `this` pointer (both explicitly and implicitly).
- **Assess** if a method should be a `const` method.
- **Write** C++ code that is const correct. (This and the last will take a lot of practice—you should start working on them, but we don't expect full mastery until the end of the semester).
- Note: we mentioned before that we aren't going to obsess over POD (and we can't even talk about all the rules for it yet). So while a professional C++ programmer should be able to **categorize** a type as POD or non-POD, we are not focused on that here.
- Note: we aren't going to make much use of static. Whatever OO language you work in, you should eventually be able to **assess** if a field or method should be static, **execute** code by hand that includes static fields and methods, and **write** code properly using static fields and methods. However, right now what we want you to do is *avoid* static until you learn more.
- **Use** the scope resolution operator to access a type declared inside another class.
- Note: later in the semester, you should be able to **write** code which properly declares types inside of other classes. In the long run, you should be able to **assess** the appropriateness of declaring a type inside of another class.
- **Select** whether a name should be a class (based on if it is a noun or a verb).
- Note there is a LOT more to OO design, and a bunch of learning objectives that will come in software engineering. We want you to start thinking about **assessing** your design choices in terms of the other ideas in 14.1.7, but we don't expect much mastery of this skill until Software Engineering.

- **References**
  - **Translate** code with references to equivalent code with a pointer.
  - **Execute** code by hand with references.
  - **Calculate** the type of an expression involving references.
  - **Write** code using references.
  - **Determine** if, when returning a reference, that reference is dangling (Note: this skill is a combination of understanding how references are implemented as pointers, and determining if a pointer you return is dangling).
  - Note: You can start thinking about **Assess** when references are the appropriate type for code—but it will take a lot of practice, well beyond this chapter. You will start by seeing a lot of situations (like operator overloading) where you should use a reference. Also related to this (and taking more practice) is **Assess** when a reference should be const—this LO is part of the LO of writing const-correct code.

- **Namespaces**
  - **Use** the scope resolution operator to use names within namespaces.

- **Function Overloading**
  - **Assess** if function overloading is appropriate, or if you should instead name the functions differently.
  - Note: We are only going to use function overloading when the resolution (picking which overload) is clear. However, if you do a lot of C++ programming beyond this course, you might need to **use** the rules of overloading resolution to determine which function will be called at a particular call site.
  - **Write** overloaded functions.
- **Operator Overloading**
  - **Assess** if operator overloading is appropriate, or if you should instead just write a method.
  - **Determine** what operator defintion (overloaded or built-in) a particular use of an operator refers to. Note that for our purposes, we are primarily concerned cases where this determination can be made from the type of the left-hand operand (*e.g.*, determining its type, including if it is a pointer or reference). Cases with more complex overloading resolution (requiring the skill noted above) are not in the scope of this course.
  - **Determine** the correct return type for a given operator (including, correctly selecting between `T`, `T &` and `const T&`).
  - **Assess** when overloadings of different `const`-ness are appropriate.
  - **Write** overloaded operators.
- **Other Aspects of Switching to C++**
  - **Use** `g++` to compile your C++ code, including appropriate options.
  - **Use** the `bool` type properly in C++ code.
  - Note: stopping use of `void *` will come in Ch 17 when you learn about templates.
  - **Analyze** separate code into `.cpp` and `.hpp` files properly.
  - **Assess** when default values are appropriate.
  - **Use** `https://cplusplus.com/reference/` for reference on the C++ library.

3 Key Learning Objectives

Let me say that we recognize there is a lot in this chapter—we need to jump from one language to another. Even though these languages are closely related[1], C++ introduces a lot of new ideas—classes, references, overloading, and many others you will see in later chapters.

If you are a bit overwhelmed, first, remember that you need to practice with these concepts before you really get them. Second, remember you can ask questions—in office hours or on the course discussion forums.

That said, here are the most key learning objectives:

[1]The very first version of C++ was "C with classes" and was compiled by a tool first converting it to C, then using the C compiler

- **Execute** code by hand with classes, method calls (including properly passing `this`), references, function overloading (where there is a clear best overloading), and operator overloading (where, after using the left hand operands type to determine whether to use the built-in operator or an overload, the best overloading is clear).

- **Write** C++ code with classes, access control, methods, references, function overloading, and operator overloading. Note that you should work on all this code being `const`-correct.

You will also want to work on **using** `https://cplusplus.com/reference/` for reference on the C++ standard library.

Note that you might want to come back and re-read this chapter after you have practiced with C++ a bit (maybe after doing excercises for this chapter and chapter 15).