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.

- **Object Creation and Destruction: Intro**
  - **Explain** the ideas of maintaining and initializing invariants, and how they relate.
  - **Explain** the problems with expecting code to call an `initialize` method whenever an object is created.
  - **Explain** what similar problem we need to solve when objects are destroyed (and why).
- **Object Creation**
  - **List** the two ways we need to make an initialize method "special".
  - **Define** *constructor*.
  - **State** the rule for how a constructor is named in C++.
  - **State** the return type of a constructor in C++.
  - **Recognize** that constructors in C++ take an implicit `this` parameter.
  - **State** the times when a constructor is (and the times when it is not) invoked.
  - **Recognize** that constructors can be overloaded (and similar rules for method overloading apply).
  - **Define** *default constructor*.
  - **State** under what conditions the compiler provides a default constructor, and how the compiler-provided default constructor behaves.
  - Note: as with Chapter 14, please don't get hung up on POD vs non-POD. This is important stuff if you do C++ professionally, but is very C++-specific.
  - **Defint** *trivial constructor*.
  - **Define** *default constructible*.
  - **State** why it is generally important to make your classes default constructible.
  - **State** the syntax for initializing an object with a constructor other than the default constructor.
  - **Recognize** that empty parenthesis at the end of a local variable declaration have a different meaning, and are not explicitly asking for the default constructor.

- **Explain** why malloc is inappropriate for allocating non-POD types (note: while we are not getting hung up on POD vs non-POD, it is still useful to know this—and in C++ we can just always use `new` which is better).
- **State** the syntax for using the `new` operator (with and without parameters to pass to the constructor).
- **Explain** how to tell the type the `new` operator evaluates to.
- **State** the syntax to use `new` to allocate an array.
- Note: the distinction between *default initialization* and *value initialization* is not something to get hung up on. If you are going to be a professional C++ programmer, you will want to learn more about that *later* (more than AoP describes). However, for now the take-away lessons are (1) always put a default constructor in classes you write and (2) a lot of people say things that are wrong (especially on the Internet).
- **Define** *initializer list*.
- **State** the syntax for an initializer list in a C++ constructor.
- **Explain** why you should use initializer lists in constructors you write.
- **Explain** why you have to initialize fields that are references or const in initializer lists.
- **State** the rule for the order in which fields are initialized in an initializer list.
- **Explain** why the order of initialization can be important.
- **List** and **explain** the four best practices in 15.1.5.

- **Object Destruction**
  - **Define** *destructor*
  - **State** the rule for how destructors are named.
  - **State** the return type for a destructor.
  - **State** the rule for the parameter list of a destructor.
  - **Explain** why some classes need destructors and others do not.
  - **Explain** why there is no equivalent of `realloc` for `new[]`.
  - Note "appreciate" is not really a valid learning objective, but we would like you to appreciate how restricted visibility helps enforce invariants.
  - **State** how to free memory allocated by `new` or `new[]`.
  - **Explain** when destructors are invoked automatically.
  - **Explain** the order of object destruction within a group of, such as an array, fields of an object, or local variables in a stack frame.
  - **Explain**, in terms of destructors, the difference between having a local variable of type `T` and a local variable of type `T*` in a stack frame, when the frame is destroyed.
  - **Explain** why you must be more careful now (in C++ than in C) about thinking about when variables go out of scope.
  - **Explain** what happens when execution reaches the close curly-brace of a destructor.
  - **Explain** what happens if you do not write any destructor in your class.

- **Define** *trivial destructor.*

- **Object Copying**
  - **List** ways that values are copied in programs. (Note: these should all be familiar, and this should be more of getting you to explicitly think about these)
  - **Explain** why naive copying is not appropriate for non-POD types.
  - **Name** the two types of copying in C++, and **explain** the difference.
  - **Explain** when the copy constructor is used.
  - **State** the parameter list for the copy constructor of a class.
  - Note: AoP mentions that you can have a non-const reference as the parameter to the copy constructor. That is for completeness, and pretty much never comes up. You can just think of the parameter as always being a const reference.
  - **Explain** what happens if you do not write your own copy constructor in a class.
  - **Explain** what it means for a copy constructor to be trivial.
  - **State** the syntax (including return type and parameter list) for overloading the copying assignment operator for a class.
  - **Explain** the three key differences in what the implementation of a copying assignment operator needs to do compared to the copy constructor.
  - **Explain** what happens if you do not write a copying assignment operator in a class.
  - **Recognize** that just seeing an `=` is not sufficient to determine that you are using the copying assignment operator: you must distinguish between initialization and assignment.
  - **Explain** how to execute code by hand with copying in C++
  - **State** the *rule of three.*
    - Note: in C++11 and later, this becomes the rule of 5.

- **Unnamed Temporaries**
  - **Define** *unnamed temporaries.*
  - **Explain** how you have seen unnamed temporaries in C, and why we did not need to explicitly discuss them.
  - **Explain** why you might need to think more carefully about unnamed temporaries in C++.
  - **Recognize** when an unnamed temporary is passed as a parameter.
  - Note: AoP mentions a subtle thing about destruction of frames—that isn't a thing to get hung up on, note that it comes with a note that it shouldn't impact what your code does, and if it does impact it you should have a really good reason and understand what is going on.
  - **Explain** why we generally prefer to have parameters that are const references instead of copying value.

- **Explain** the copying and destruction that takes place when a value is returned, and that return value is used.
- **Define** *return value optimization*.
- Note: we aren't going to require that you do anything with return value optimization explicitly, but you should understand it exists—especially in terms of not getting confused with debugging your code.
- **Recall** *type conversion* (from Chapter 3).
- **State** that in C++ one-parameter constructor can be used for implicit type conversion, unless declared `explicit`.
- **Explain** why you should declare all one parameter constructors, except the copy constructor as `explicit`. **Note:** you should aim to get into this habit right away!
- **Explain** how overloaded function make non-explicit constructors (or other implicit conversions) particularly dangerous.

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 Creation**
  - **Execute** by hand code which uses constructors, initializer lists, classes that have fields which have constructors, `new`, and/or `new[]`.
  - **Write** constructors which properly initialize your C++ classes, making use of initialize lists where appropriate (which is for all initialization).
  - **Write** code which uses `new` and `new[]` to dynamically allocate memory.
    - Note: you should stop using `malloc` in C++ code.
  - **Follow** the 4 best practices laid out in 15.1.5 (Make classes default constructible, Use initializer lists to initialize your class's fields, In the initializer list explicitly initialize every field, Initialize fields in the order they are declared).
- **Object Destruction**
  - **Execute** code by hand which uses destructors, including determining where destructors are invoked (*e.g.*, local variables in a frame being destroyed, fields in an object at the end of its destructor, when using `delete`/`delete[]`).
  - **Determine** when lack of a destructor or a problem with a destructor causes (or could cause) memory-related errors (such as memory leaks, double frees, etc).
  - **Write** destructors for classes that properly free resources (such as deleting dynamically allocated memory).
- **Object Copying**

  - **Execute** code by hand which includes copy constructors and copying assignment operators (including correctly selecting whether the copy constructor or assignment operator is used in a particular situation, copying parameters and return values appropriately, etc).
  - **Determine** when a class requires *Rule of Three* methods.
  - **Write** copy constructors and copying assignment operators for classes that properly copy (usually deep) the objects. The copying assignment operator should also appropriately free resources, check for self assignment, and return `*this`.

- **Unnamed Temporaries**
  - **Execute** code by hand which includes the use of unnamed temporaries. Note: most of the time this doesn't require much special, and we won't focus on the cases where special handling is needed. However, if you write such in your own code, understanding the behavior is likely to help you sort out problems.
  - **Declare** one parameter constructors (other than the copy constructor) as `explicit`.
    * Note: the more advanced LO of "**Assess** when a one parameter constructor should not be declared `explicit`" is significantly outside the scope of this class—just make them all explicit.
  - **Use** the best practice of avoiding implicit conversions.

3 Key Learning Objectives

Our most key learning objectives are to **execute** code by hand and **write** code using the ideas in this chapter (as detailed in the second reading learning objectives). A key part of writing the your own classes correctly is **determining** when a class requires *Rule of Three* methods.

Furthermore, we want to note that understanding the Rule of Three (or in C++11 and later, the Rule of Five—See Appendix E.6.4 to reading about move constructors and move assignment operators, which are NOT a part of the core content of this class) is a key indicator of someone who is serious about C++, as opposed to *e.g.*, a C programmer using a few C++ features.

We also want you to start on the best practices outlined in this chapter (the 4 in 15.1.5, and using `explicit`) now, so they become ingrained.