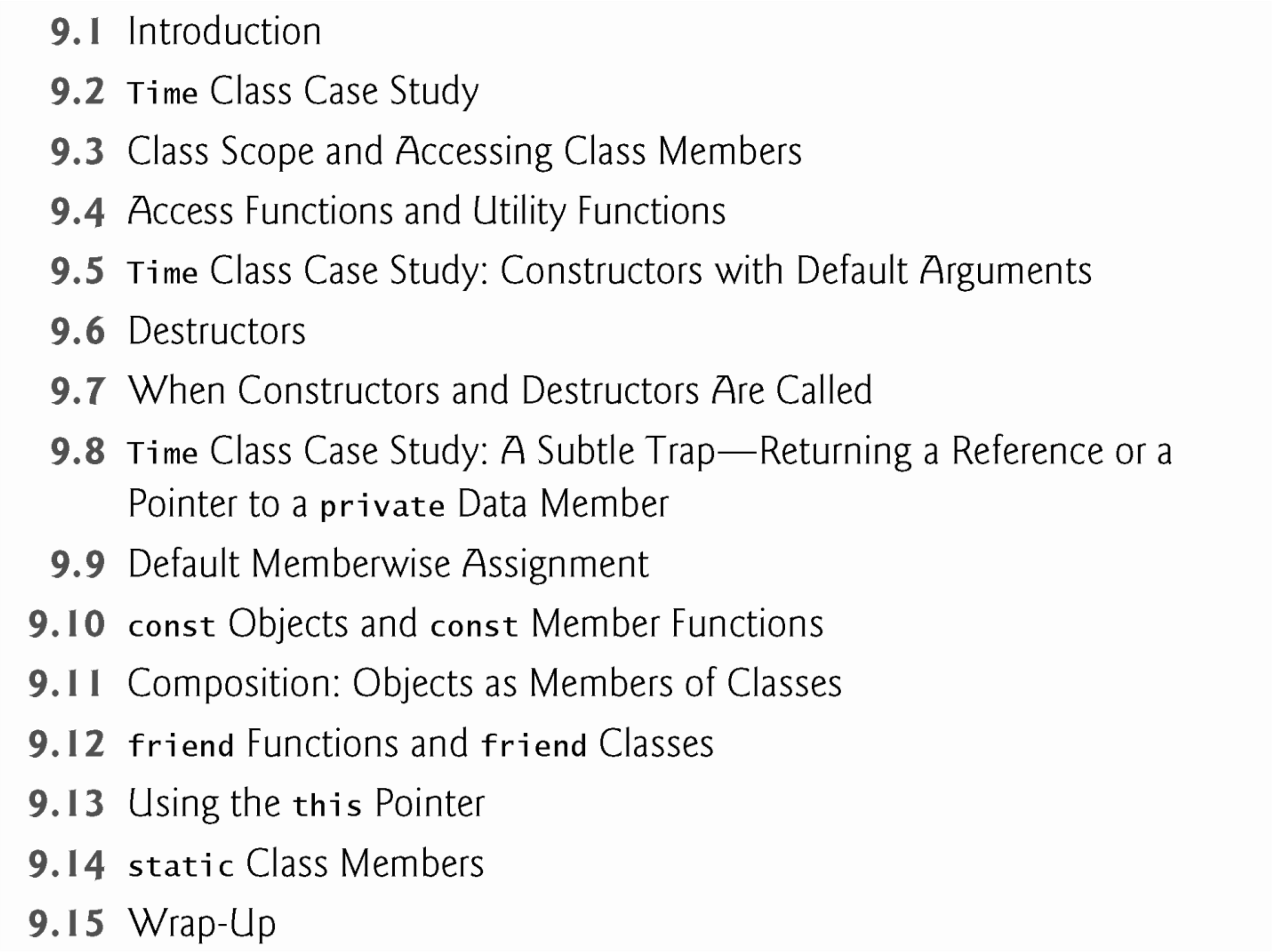
Chapter 9
Classes: A Deeper Look; Throwing Exceptions

C++ How to Program, 9/e

OBJECTIVES

In this chapter you'll:

- Use an include guard.
- Access class members via an object's name, a reference or a pointer.
- Use destructors to perform "termination housekeeping."
- Learn the order of constructor and destructor calls.
- Learn about the dangers of returning a reference to `private` data.
- Assign the data members of one object to those of another object.
- Create objects composed of other objects.
- Use `friend` functions and `friend` classes.
- Use the `this` pointer in a member function to access a non-`static` class member.
- Use `static` data members and member functions.



9.1 Introduction

9.2 `Time` Class Case Study

9.3 Class Scope and Accessing Class Members

9.4 Access Functions and Utility Functions

9.5 `Time` Class Case Study: Constructors with Default Arguments

9.6 Destructors

9.7 When Constructors and Destructors Are Called

9.8 `Time` Class Case Study: A Subtle Trap—Returning a Reference or a Pointer to a `private` Data Member

9.9 Default Memberwise Assignment

9.10 `const` Objects and `const` Member Functions

9.11 Composition: Objects as Members of Classes

9.12 `friend` Functions and `friend` Classes

9.13 Using the `this` Pointer

9.14 `static` Class Members

9.15 Wrap-Up

9.1 Introduction

- This chapter takes a deeper look at classes.
- Coverage includes:
  - The example also demonstrates using an *include guard* in headers to prevent header code from being included in the same source code file more than once.
  - We demonstrate how client code can access a class's `public` members via the name of an object, a reference to an object or a pointer to an object.
  - We discuss access functions that can read or write an object's data members.

9.1 Introduction (cont.)

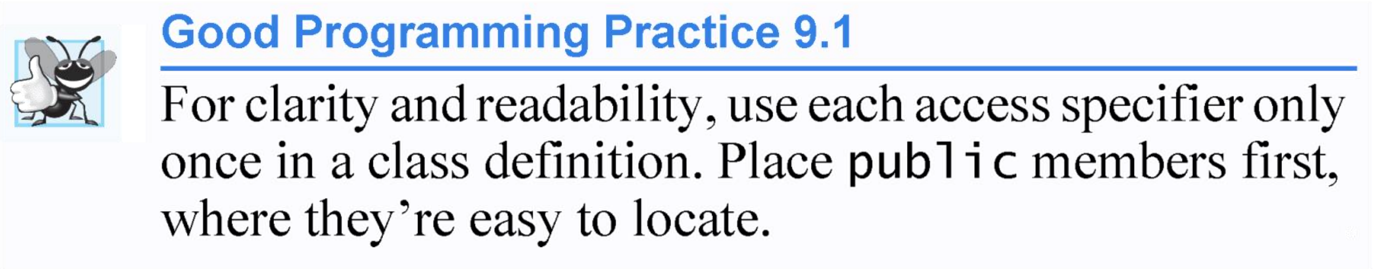
- Coverage includes (cont.):
  - How default arguments can be used in constructors.
  - Destructors that perform "termination housekeeping" on objects before they're destroyed.
  - The *order* in which constructors and destructors are called.
  - We show that returning a reference or pointer to `private` data *breaks the encapsulation* of a class, allowing client code to directly access an object's data.

9.1 Introduction (cont.)

- Coverage includes (cont.):
  - `const` objects and `const` member functions to prevent modifications of objects and enforce the principle of least privilege.
  - *Composition*—a form of reuse in which a class can have objects of other classes as members.
  - *Friendship* to specify that a nonmember function can also access a class's non-public members—a technique that's often used in operator overloading for performance reasons.
  - `this` pointer, which is an implicit argument in all calls to a class's non-static member functions,

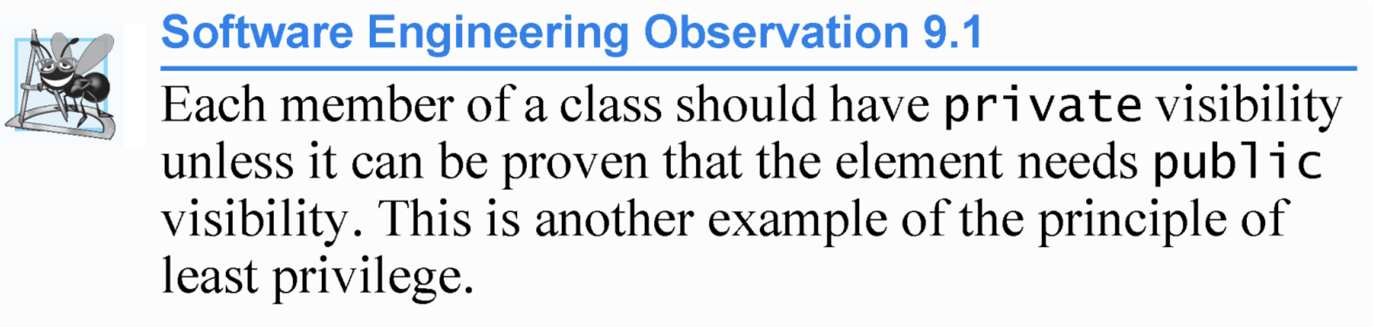
9.2 `Time` Class Case Study

- Our first example (Fig. 9.1) creates class `Time` and tests the class.



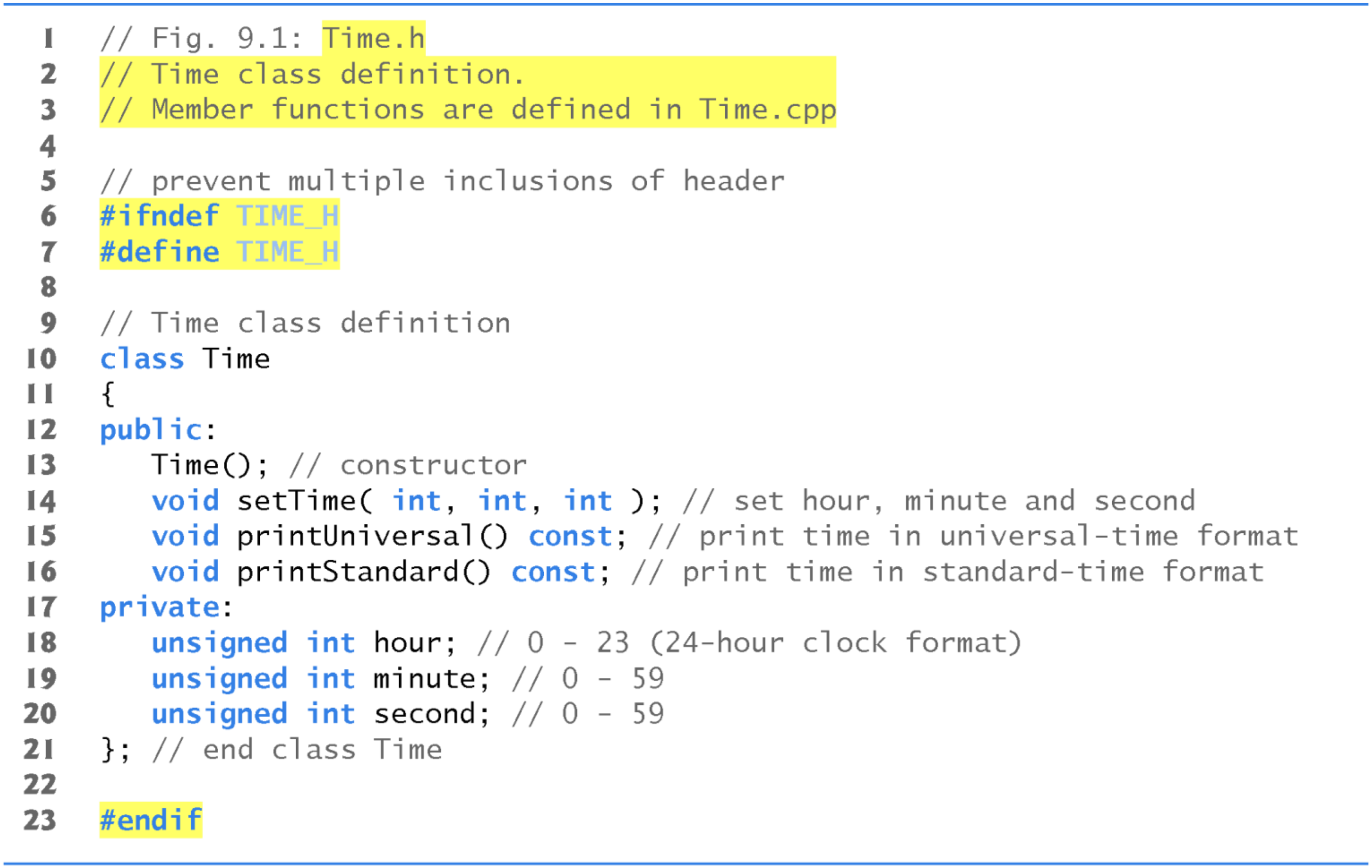
Good Programming Practice 9.1

For clarity and readability, use each access specifier only once in a class definition. Place `public` members first, where they’re easy to locate.



Software Engineering Observation 9.1

Each member of a class should have `private` visibility unless it can be proven that the element needs `public` visibility. This is another example of the principle of least privilege.



```
// Fig. 9.1: Time.h
// Time class definition.
// Member functions are defined in Time.cpp

// prevent multiple inclusions of header
#ifndef TIME_H
#define TIME_H

// Time class definition
class Time
{
public:
   Time(); // constructor
   void setTime( int, int, int ); // set hour, minute and second
   void printUniversal() const; // print time in universal-time format
   void printStandard() const; // print time in standard-time format
private:
   unsigned int hour; // 0 - 23 (24-hour clock format)
   unsigned int minute; // 0 - 59
   unsigned int second; // 0 - 59
}; // end class Time

#endif
```

Fig. 9.1 | Time class definition.

9.2 Time Class Case Study (cont.)

- In Fig. 9.1, the class definition is enclosed in the following include guard:

```
// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H
   ...
#endif
```

  - Prevents the code between `#ifndef` and `#endif` from being included if the name `TIME_H` has been defined.
  - If the header has *not* been included previously in a file, the name `TIME_H` is *defined* by the `#define` directive and the header file statements are included.
  - If the header has been included previously, `TIME_H` is defined already and the header file is not included again.

Error-Prevention Tip 9.1

Use `#ifndef`, `#define` and `#endif` preprocessing directives to form an include guard that prevents headers from being included more than once in a source-code file.

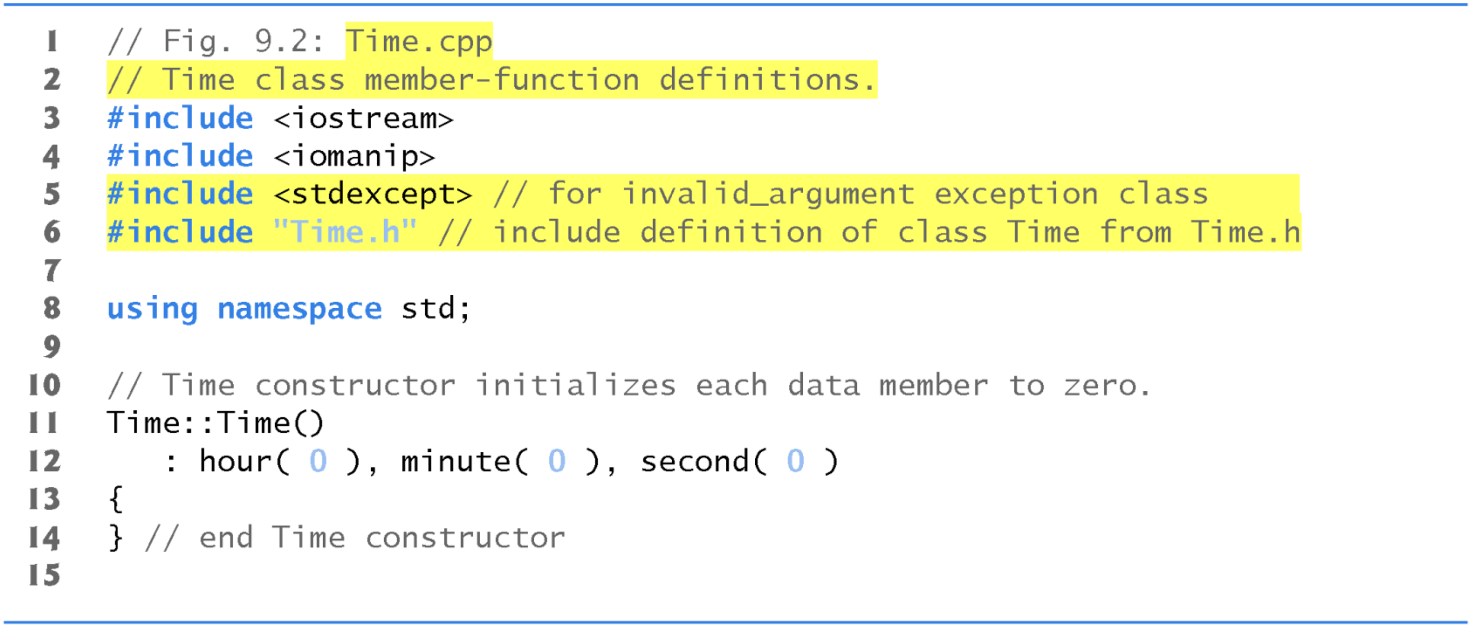
Good Programming Practice 9.2

By convention, use the name of the header in uppercase with the period replaced by an underscore in the `#ifndef` and `#define` preprocessing directives of a header.

9.2 `Time` Class Case Study (cont.)

***Time* Class Member Functions**

- In Fig. 9.2, the `Time` constructor (lines 11–14) initializes the data members to 0—the universal-time equivalent of 12 AM.
- Invalid values cannot be stored in the data members of a `Time` object, because the constructor is called when the `Time` object is created, and all subsequent attempts by a client to modify the data members are scrutinized by function `setTime` (discussed shortly).
- You can define *overloaded constructors* for a class.



```cpp
// Fig. 9.2: Time.cpp
// Time class member-function definitions.
#include <iostream>
#include <iomanip>
#include <stdexcept> // for invalid_argument exception class
#include "Time.h" // include definition of class Time from Time.h

using namespace std;

// Time constructor initializes each data member to zero.
Time::Time()
   : hour( 0 ), minute( 0 ), second( 0 )
{
} // end Time constructor

```

Fig. 9.2 | Time class member-function definitions. (Part 1 of 3.)