Good Programming Practice 9.3

For clarity, list member initializers in the order that the class's data members are declared.

9.11 Composition: Objects as Members of Classes (cont.)

`Date` Class's Default Copy Constructor

- As we mentioned in Section 9.9, the compiler provides each class with a *default copy constructor* that copies each data member of the constructor's argument object into the corresponding member of the object being initialized.
- Chapter 10 discusses how you can define customized copy constructors.

9.11 Composition: Objects as Members of Classes (cont.)

Testing Classes `Date` and `Employee`

- Figure 9.21 creates two `Date` objects (lines 10–11) and passes them as arguments to the constructor of the `Employee` object created in line 12.
- Line 15 outputs the `Employee` object's data.
- When each `Date` object is created in lines 10–11, the `Date` constructor defined in lines 11–25 of Fig. 9.18 displays a line of output to show that the constructor was called (see the first two lines of the sample output).

9.11 Composition: Objects as Members of Classes (cont.)

- [Note: Line 12 of Fig. 9.21 causes two additional Date constructor calls that do not appear in the program's output. When each of the Employee's Date member objects is initialized in the Employee constructor's member-initializer list (Fig. 9.20, lines 14–15), the default copy constructor for class Date is called. Since this constructor is defined implicitly by the compiler, it does not contain any output statements to demonstrate when it's called.]

```
1  // Fig. 9.21: fig09_21.cpp
2  // Demonstrating composition--an object with member objects.
3  #include <iostream>
4  #include "Date.h" // Date class definition
5  #include "Employee.h" // Employee class definition
6  using namespace std;
7
8  int main()
9  {
10    Date birth( 7, 24, 1949 );
11    Date hire( 3, 12, 1988 );
12    Employee manager( "Bob", "Blue", birth, hire );
13
14    cout << endl;
15    manager.print();
16 } // end main
```

Fig. 9.21 | Demonstrating composition—an object with member objects. (Part 1 of 2.)

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

There are actually five constructor calls when an `Employee` is constructed—two calls to the `string` class's constructor (lines 12–13 of Fig. 9.20), two calls to the `Date` class's default copy constructor (lines 14–15 of Fig. 9.20) and

Fig. 9.21 | Demonstrating composition—an object with member objects. (Part 2 of 2.)

9.11 Composition: Objects as Members of Classes (cont.)

What Happens When You Do Not Use the Member Initializer List?

- If a member object is not initialized through a member initializer, the member object's *default constructor* will be called *implicitly*.
- Values, if any, established by the default constructor can be overridden by set functions.
- However, for complex initialization, this approach may require significant additional work and time.

Common Programming Error 9.5

A compilation error occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).

Performance Tip 9.4

Initialize member objects explicitly through member initializers. This eliminates the overhead of "doubly initializing" member objects—once when the member object's default constructor is called and again when set functions are called in the constructor body (or later) to initialize the member object.

Software Engineering Observation 9.11

If a data member is an object of another class, making that member object `public` does not violate the encapsulation and hiding of that member object's `private` members. But, it does violate the encapsulation and hiding of the containing class's implementation, so member objects of class types should still be `private`.

9.12 `friend` Functions and `friend` Classes

- A friend function of a class is a non-member function that has the right to access the `public` *and* `non-public` class members.
- Standalone functions, entire classes or member functions of other classes may be declared to be *friends* of another class.

9.12 `friend` Functions and `friend` Classes (cont.)

Declaring a friend

- To declare a function as a `friend` of a class, precede the function prototype in the class definition with keyword `friend`.
- To declare all member functions of class `ClassTwo` as friends of class `ClassOne`, place a declaration of the form

```
friend class ClassTwo;
```

- in the definition of class `ClassOne`.
- Friendship is *granted*, *not taken*—for class B to be a `friend` of class A, class A *must* explicitly declare that class B is its `friend`.
- Friendship is not *symmetric*—if class A is a friend of class B, you cannot infer that class B is a friend of class A.
- Friendship is not *transitive*—if class A is a friend of class B and class B is a friend of class C, you cannot infer that class A is a friend of class C.

9.12 `friend` Functions and `friend` Classes (cont.)

Modifying a Class's `private` Data with a Friend Function

- Figure 9.22 is a mechanical example in which we define `friend` function `setX` to set the private data member `x` of class `Count`.
- We place the `friend` declaration *first* in the class definition, even before `public` member functions are declared.
- Function `setX` is a stand-alone (global) function—it isn't a member function of class `Count`.
- For this reason, when `setX` is invoked for object counter, line 41 passes counter as an argument to `setX` rather than using a handle (such as the name of the object) to call the function, as in

```
counter.setX( 8 ); // error: setX not a member function
```

- If you remove the friend declaration in line 9, you'll receive error messages indicating that function `setX` cannot modify class `Count`'s `private` data member `x`.

```
1  //Fig. 9.22: fig09_22.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using namespace std;
5
6  // Count class definition
7  class Count
8  {
9     friend void setX( Count &, int ); // friend declaration
10 public:
11    // constructor
12    Count()
13       : x( 0 ) // initialize x to 0
14    {
15       // empty body
16    } // end constructor Count
17
```

Fig. 9.22 | Friends can access `private` members of a class. (Part 1 of 3.)

```
18      // output x
19      void print() const
20      {
21         cout << x << endl;
22      } // end function print
23   private:
24      int x; // data member
25   }; // end class Count
26
27   // function setX can modify private data of Count
28   // because setX is declared as a friend of Count (line 9)
29   void setX( Count &c, int val )
30   {
31      c.x = val; // allowed because setX is a friend of Count
32   } // end function setX
33
```

Fig. 9.22 | Friends can access `private` members of a class. (Part 2 of 3.)